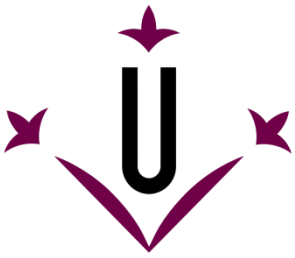
UNIVERSITAT DE LLEIDA
Escola Politècnica Superior



Processament i visualització d'un stream de dades amb Clojure i ClojureScript

Treball final de grau

Albert Berga Gatius
Tutoritzat per: Juan Manuel Gimeno Illa
Juny de 2015

CONTINGUTS

ÍNDEX DE TAULES

ÍNDEX DE FIGURES

ÍNDEX DE FRAGMENTS DE CODI

1 INTRODUCCIÓ

En aquest treball desenvoluparem una aplicació web que analitzarà l'*stream* de Twitter per extreure estadístiques dels *tweets* i mostrar-les gràficament en una pàgina web conjuntament amb els *tweets* que van arribant a l'aplicació. Els llenguatges que utilitzarem per desenvolupar l'aplicació seran Clojure, per la part del servidor, i ClojureScript, per al client. Ambdós són llenguatges de programació funcional totalment nous per a mi, així com també ho és la programació funcional. És per aquest motiu que, per poder desenvolupar el projecte, caldrà fer un gran esforç inicial d'aprenentatge del llenguatge i de les eines relacionades amb aquest [A2].

La motivació principal que ens ha conduit a escollir aquest tema pel treball ha estat la de voler veure quelcom diferent al que estàvem acostumats durant el transcurs del grau. Allunyant-nos de la programació orientada a objectes per introduir-nos en la programació funcional. Cal destacar que, inicialment, estàvem buscant un treball per programar orientat a objectes amb Java. No obstant això, vaig preguntar al tutor per la programació funcional i em va agradar la idea. Així que ens vam aventurar en aquest treball.

D'altra banda, la decisió d'analitzar l'*stream* de Twitter la vam prendre degut a que coneixíem l'existència de la API de Twitter [W8], la qual permet rebre l'*stream* de *tweets*. Vam creure que podríem fer alguna cosa interessant analitzant el que rebíem per tal d'extreure algun tipus d'estadística. En un principi no teníem del tot clar les possibilitats que donava la API de Twitter ni el que podríem fer amb ella. No obstant això, vam tirar endavant el projecte creient que podríem fer alguna cosa interessant.

Fent referència a la memòria del projecte, l'hem estructurat en tres parts ben diferenciades. En la primera d'elles (**FASE PRÈVIA (APRENENTATGE)**) fem una explicació de cadascun dels conceptes sobre els quals hem fet un aprenentatge previ al desenvolupament del projecte. En la segona part de la memòria hem planificat el desenvolupament del projecte, hem definit uns requeriments que intentarem satisfer els hem dividit en diferents iteracions (**ANÀLISI DE REQUERIMENTS**). D'aquesta forma hem

planificat el projecte en petits passos que ens resulten en versions estables del projecte. Finalment, l'últim apartat de la memòria (**DESENVOLUPAMENT**), explica els avanços realitzats en cadascuna de les iteracions, tot detallant la implementació del codi tant en la part del servidor com en la del client.

Cal destacar que hem diferenciat les referències bibliogràfiques en diferents apartats, ja que per realitzar el treball hem realitzar recursos de diferents característiques. És per això que al llarg del treball trobarem referències a les cites bibliogràfiques en una nomenclatura una mica diferent a l'habitual. A banda dels llibres, pels quals utilitzem la nomenclatura normal [número], tenim articles (A), presentacions (P), repositoris GitHub (R) i pàgines web (W), pels quals afegim davant del número de cita la lletra diferenciadora dels diferents apartats.

2 FASE PRÈVIA (APRENENTATGE)

2.1 Clojure

2.1.1 Visió global del llenguatge

Clojure [2] [3] és un llenguatge de programació funcional de propòsit general i dialecte de Lisp. Fou dissenyat per Rich Hickey i es va publicar l'any 2007. Segons Hickey, va dissenyar Clojure en el seu intent de trobar un llenguatge que no va poder trobar: un Lisp funcional integrat sobre un entorn robust i amb la programació concurrent en ment. El llenguatge pot ser executat sobre la JVM (Java Virtual Machine) i sobre la màquina virtual de la plataforma .NET. A més, també pot ser compilat a JavaScript per ser executat en un navegador, cobrint així la part del client en una aplicació web. En aquest cas, el llenguatge s'anomena ClojureScript. Tractarem aquest tema en detall més endavant.

2.1.2 Característiques

2.1.2.1 Sintaxi

El primer que cal observar és la forma com en Clojure es representen els diferents tipus de dades, els quals queden descrits en la següent taula:

Tipus	**Exemple**
cadena de caràcters	"foo"
caràcter	\f
enter	42

punt flotant	3.14
boleà	true
nil	nil
símbol	+, -, ...
paraula clau[1]	:foo

Taula 1 Tipus de dades Clojure [P2]

En segon lloc, cal veure les diferents estructures de dades que podem utilitzar en Clojure, de les quals veurem en més detall el seu comportament en la secció **Estructures de dades immutables i persistents.**

Tipus	**Propietats**	**Exemple**
Llista	seqüencial	(1 2 3)
Vector	seqüencial i d'accés aleatori	[1 2 3]
Taula associativa	associativa (clau-valor)	{:a 100 :b 90}
Set	no repetició	#{:a :b}

Taula 2 Tipus d'estructures de dades Clojure [P2]

Quant a la sintaxi del codi, com la resta de llenguatges Lisp la sintaxi està construïda sobre expressions simbòliques que són convertides en estructures de dades per un lector abans de ser compilades. Les expressions estan delimitades per parèntesis y amb notació prefixa, amb el qual es crida al primer membre de cada llista com a funció,

[1] No confondre amb paraules clau del llenguatge de programació.

passant la resta d'elements com a arguments d'aquesta. A més a més, tal i com podem veure en la següent taula, tots els tipus d'expressions tenen la mateixa forma.

Tipus	**Exemple**
Funció	(println "hello")
Operador	(+ 1 2)
Crida a mètode	(.trim " hello ")
Import	(require 'mylib)
Metadata	(with-meta obj m)
Control del flux	(when valid? (proceed))
Abast	(dosync (alter ...))

Taula 3 Tipus d'expressions Clojure [P2]

D'altra banda, Clojure també permet definir funcions pròpies igual que ho podríem fer en qualsevol altre llenguatge de programació. La manera de definir una funció és la següent:

```
(defn nom-funció [arg1 arg2 ...]
  (cos-funció))
```

Codi 2.1-1 Exemple funció Clojure

A més a més, també disposa de la possibilitat de definir una funció que actuï diferent en funció del número d'arguments que se li passin.

```
(defn saluda
  ([] (saluda "World"))
  ([name] (println "Hello " name)))
(saluda) => Hello  World
(saluda "Albert") => Hello Albert
```

Codi 2.1-2 Exemple funció n-ària

També existeix la possibilitat de no definir exactament el número d'arguments que esperem rebre en una funció, de manera que rebrem els arguments en forma de seqüència.

```
(defn n-args [& args]
  args)
(n-args "a" "b" "c" 1 2)
=> ("a" "b" "c" 1 2)
```

Codi 2.1-3 Exemple funció que rep paràmetres en forma de seqüència

Finalment, una característica molt útil de Clojure és la "desestructuració". Aquesta permet vincular valors a símbols i descompondre una estructura de dades per obtenir de forma directa les seves parts. S'utilitza bàsicament en rebre paràmetres en funcions o en sentències "let", veurem un exemple d'aquest tipus:

```
(let [[n11 n12 n13 n14] [11 12 13]          ;n11 = 11, n12 = 12,
                                             n13 = 13, n14 = nil
      [n21 & other] [21 22 23]              ;n21 = 21, other = (22, 23)
      [n31 n32 [n33 n34]] [31 32 [33 34]]   ;n31 = 31, n32 = 32,
                                             n33 = 33, n34 = 34
      {:keys [a b c]} {:a 41 :b 42 :c 43}]) ;a = 41, b = 42, c = 43
```

Codi 2.1-4 Exemples desestructuració

Podem veure que en cada cas podem obtenir directament els elements continguts en les estructures de dades, obtenint en cada símbol el valor que es mostra en la part

dreta en forma de comentari. Cal destacar que, com en el primer cas, si l'estructura de dades no té suficients valors per vincular amb els símbols, aquests tindran valor `nil`.

2.1.2.2 Estructures de dades immutables i persistents

En Clojure, les estructures de dades són immutables. És a dir, el seu valor no pot canviar i alhora conservar la instància de l'estructura de dades. Per exemple, si tenim una llista amb tres valors i volem afegir un quart, es crearà una nova llista que serà una instància diferent de la que teníem amb els tres elements. No obstant això, aquests elements seran els mateixos en la nova llista i no una còpia. Amb la qual cosa tindrem tres elements compartits en les dues llistes però no suposarà cap problema, donat que aquests elements són immutables. És per això que diem que les estructures de dades en Clojure són persistents [A5], perquè els elements ja existents es mantenen en modificar una estructura de dades.

Per entendre millor aquest mecanisme ho veurem mitjançant un exemple d'implementació d'un arbre binari de cerca, el qual hem extret de la presentació del Juan Manuel Gimeno "Functional programming in Clojure" [P1]. En aquest exemple, cada node de l'arbre tindrà tres camps; el valor del node, un apuntador al fill dret i un a l'esquerra. En cas de representar un arbre buit ho farem mitjançant la constant `nil`. A més a més, implementarem la funció "xconj", la qual afegirà un nou valor a l'arbre.

```
(defn xconj [t v]
 (cond
   (nil? t)
     {:L nil    :val v    :R nil}
   (< v (:val v))
     {:L (xconj (:L t) v)    :val (:val t)    :R (:R t)}
   :else
     {:L (:L t)    :val (:val t)    :R (xconj (:R t) v)}))
```

Codi 2.1-5 Funció d'inserció en arbre binari de cerca Clojure

Podem observar que en cridar la funció anterior per introduir un nou valor en l'arbre, aquesta es cridarà recursivament anant descendint pels nodes de l'arbre fins a trobar el lloc on col·locar el valor. El que aquí ens interessa veure és com en modificar aquest arbre per introduir un nou valor, la part de l'arbre que no ha estat modificada no canvia.

```
(def tree1
  (-> nil
      (xconj 5)
      (xconj 3)
      (xconj 2)))

(def tree2 (xconj tree1 7))
```

Codi 2.1-6 Exemple creació arbres binaris de cerca amb funció xconj

En l'exemple anterior creem un primer arbre amb tres valors i posteriorment en creem un altre a partir d'afegir un nou valor a aquest. Com el nou valor que afegim en crear el segon arbre és major que tots els altres s'afegirà en el fill dret de l'arrel i, per tant, el fill esquerra no haurà canviat. És el que comprovem mitjançant la funció `identical?` i la qual ens retorna `true`. El que està passant en aquest arbre és el següent:

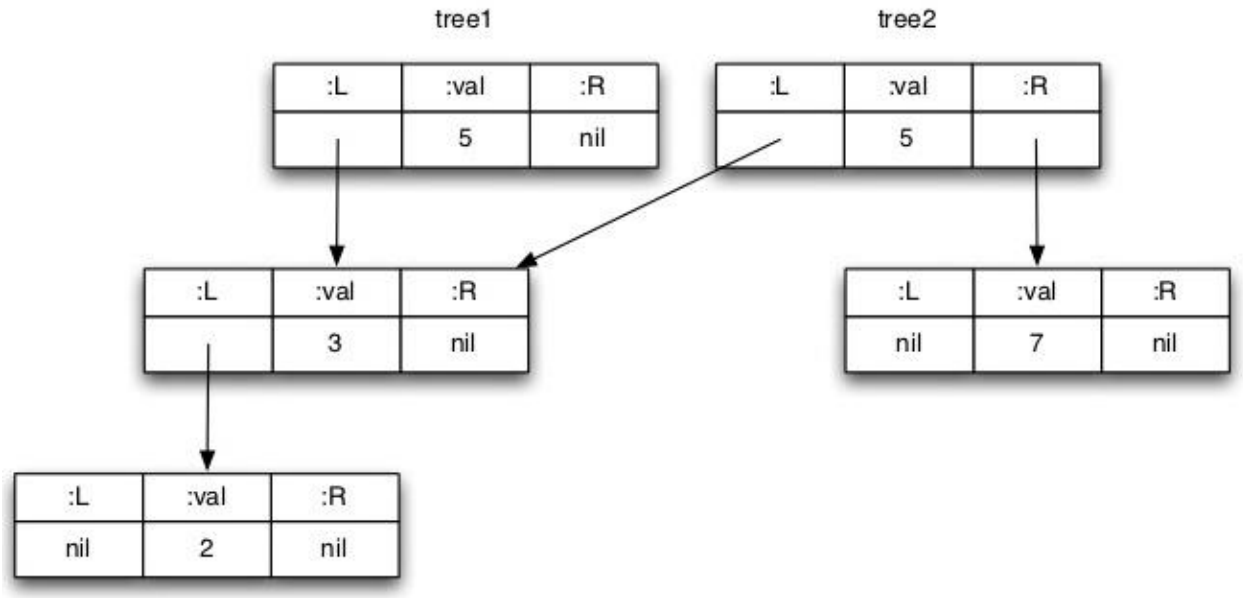


Figura 1 Representació inserció en arbre binari de cerca en Clojure

Podem veure que la part comú en els dos arbres és compartida. La qual cosa no és perillosa degut a que els nodes són immutables i no poden canviar. És per això que el node arrel de l'arbre no s'ha pogut aprofitar, ja que en ser immutable no pot canviar el valor de `:R` per apuntar al nou node.

Clojure guarda els valors de les estructures de dades de forma similar a l'exemple que hem vist. En aquest cas, l'arbre no és binari sinó que cada node pot tenir fins a 32 fills i l'arbre pot tenir un màxim de 7 nivells de profunditat. Això és degut a que la forma com Clojure determina en quina posició de l'arbre situar un valor és mitjançant un codi binari de 32 bits, mitjançant els quals es determina la posició del nou valor. Aquest codi binari correspon a la posició del valor, en cas de ser un vector, o al codi de `hash`, en cas de ser una taula associativa. Per determinar la posició a la que correspon un d'aquests codis es calcula de la següent forma:

Cada node de l'arbre conté 5 bits, els quals serveixen per determinar el node en el qual s'emmagatzemarà el valor en el nivell inferior. Es repeteix aquest procés fins a acabar els 30 primers bits del codi. Els dos restants serveixen per determinar quina posició del node fulla en el que hem arribat es correspon amb el codi, ja que cada fulla té 8 posicions. En la següent imatge podem veure un exemple de l'arbre corresponent a una taula associativa.

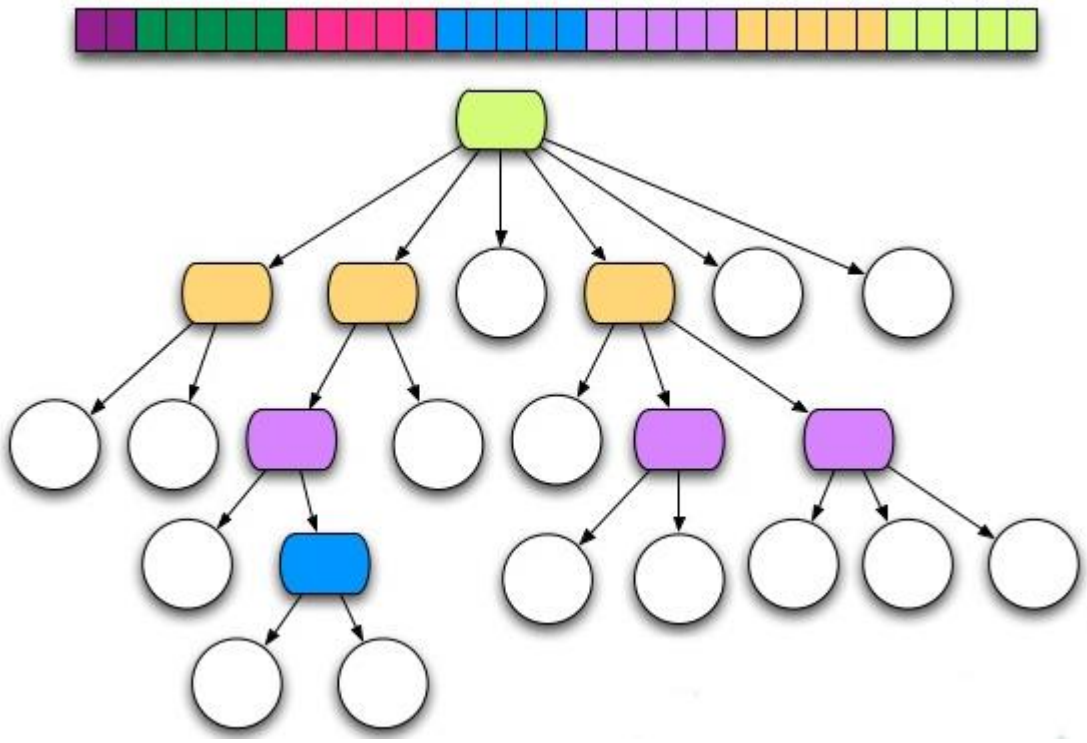


Figura 2 Representació taula associativa Clojure [P3]

Es pot observar com en cada nivell s'utilitzen els 5 següents bits del codi. A més a més, les rodones blanques representen fulles de l'arbre. Sorprèn que ja tinguem fulles en el segon nivell de l'arbre, la qual cosa es dóna degut a que en cas que la branca de l'arbre només contingui una fulla, no es representen els nodes intermitjos.

A continuació veurem un exemple [A5] de les possibles situacions que es poden donar en introduir un nou valor en l'estructura de dades en forma d'arbre. Per simplificar-ho, els arbres d'exemple seran binaris. I, ja que es tracta d'un vector, seran complerts per l'esquerra. En canvi, a la figura anterior, que es correspon a una taula associativa, no tenim aquesta regularitat. Aquí les claus són les posicions del vector que sempre són 0, 1, 2, ...

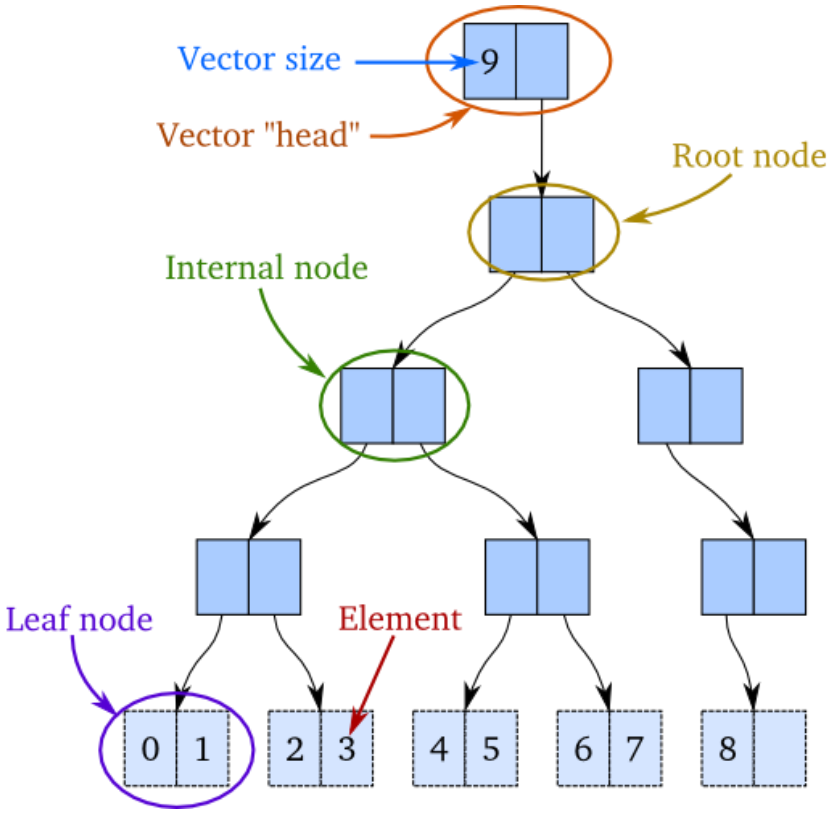


Figura 3 Arbre que representa vector Clojure simplificat

Cal tenir en compte que els diferents nodes són immutables i, per tant, si volem afegir un nou número al final d'aquesta llista no podrem afegir-lo directament en una fulla que té un espai lliure, sinó que haurem de crear un node nou. A més a més, no sol haurem de canviar el node fulla, sinó tots els nodes ascendents d'aquest per tal de que apuntin al nou node. Aquí en tenim un exemple en el qual, a més d'haver d'afegir una nova fulla, és necessari incrementar en un nivell l'arbre per tal de poder referenciar totes les fulles.

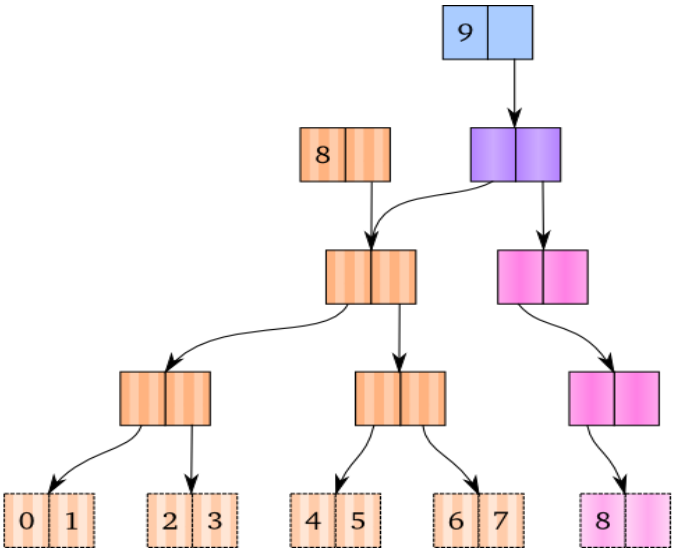


Figura 4 Representació d'inserció en arbre que representa vector Clojure. Cas 1

En cas que la operació que haguem de fer sigui modificar algun valor, la idea seria la mateixa. Afegiríem un nou node per a la fulla on estigui el valor a modificar, amb la conseqüència d'haver de canviar també tots els nodes ascendents.

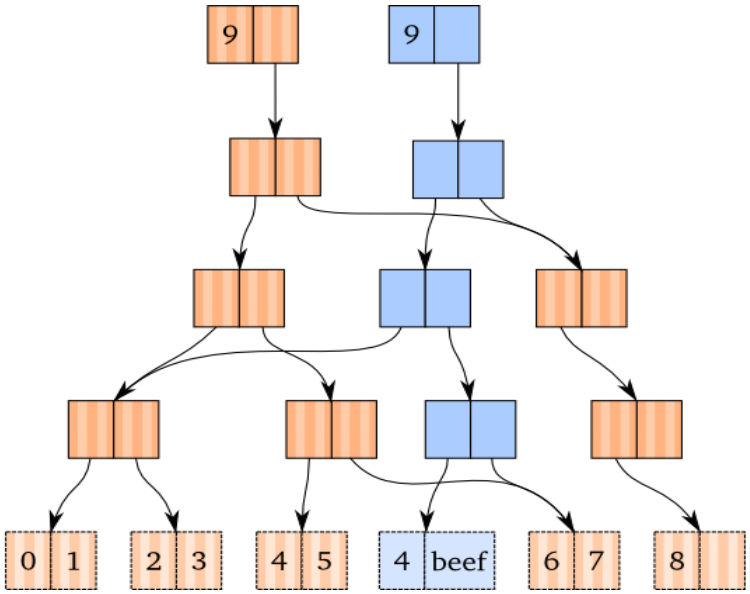


Figura 5 Representació d'inserció en arbre que representa vector Clojure. Cas 2

Contràriament a l'exemple que hem vist per al cas d'afegir un nou element, en el cas de voler eliminar un element hauríem de fer el pas invers. Per tant, ara haurem de reduir en un nivell l'arbre i deixar fora d'aquest la part que ens sobra.

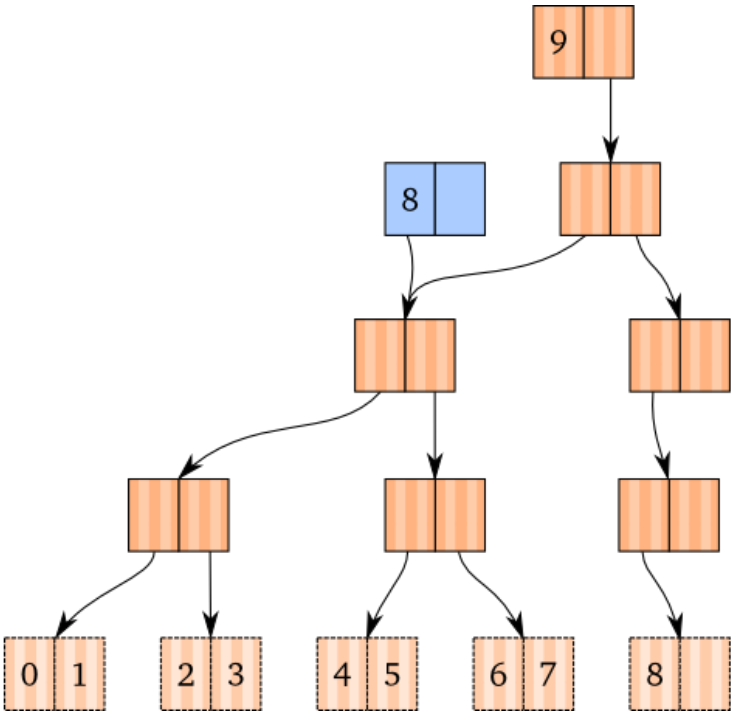


Figura 6 Representació d'esborrat en arbre que representa vector Clojure

En els exemples anteriors hem pogut veure com es comportarien els vectors en aplicar-los les diferents operacions que tenim. No obstant això, hi ha altres situacions en les quals la modificació de l'arbre seria una mica diferent respecte al que hem vist. Tot i això, el que ens interessa veure és la idea, ja que la resta de possibles situacions es basen en la mateixa idea.

2.1.2.3 Concurrència

Tal com hem esmentat anteriorment, Clojure va ser creat pensant en la programació concurrent i, per tant, té algunes peculiaritats que afavoreixen aquest tipus de programació.

En primer lloc, com hem explicat en l'apartat anterior, les estructures de dades són immutables, és a dir, no poden canviar de valor. Si volem canviar el valor d'una estructura obtindrem una nova estructura amb el nou valor. D'aquesta forma ens assegurem que el nostre codi no tindrà efectes secundaris, especialment quan s'executin trossos de codi concurrentment que operin sobre les mateixes estructures de dades.

En segon lloc, les funcions són pures. S'entén que una funció és pura si:

- Donats uns paràmetres d'entrada sempre es retorna la mateixa sortida.
- No causa cap efecte col·lateral. No canvia el món extern, cap variable externa mutable ni genera cap sortida per entrada/sortida.

D'aquesta forma ens assegurem que al cridar la funció no canviarà res que pugui afectar a altres parts del nostre programa. Tot i això, donat que Clojure no és un llenguatge funcional pur (p.e. Haskell) permet crear fàcilment excepcions respecte al que hem comentat. Ni totes les estructures de dades són immutables, ni totes les funcions són pures. No obstant això, quan hem d'incomplir alguna d'aquestes peculiaritats haurem de fer-ho de forma controlada.

A banda d'aquestes característiques que afavoreixen la concurrència, Clojure disposa d'altres característiques dissenyades expressament per afavorir el desenvolupament de programes concurrents. Tot i això, en aquest treball només parlarem de l'àtom, el qual serà necessari per desenvolupar la nostra aplicació.

2.1.2.4 Àtom

En Clojure, un àtom és un tipus de referència que es comporta de forma semblant a com ho fa una estructura de dades mutable. Un àtom és una referència a una estructura de dades de Clojure, la qual és immutable. No obstant això, podem dir que un àtom es comporta com una estructura de dades mutable degut a que podem canviar l'estructura de dades a la que apunta la referència de l'àtom sense que variï la instància de l'àtom.

Els àtoms de Clojure disposen de funcions per canviar el contingut d'aquests, o millor dit, canviar l'estructura a la que fa referència l'àtom. D'aquesta forma, tot i estar treballant amb una estructura "mutable" evitarem al màxim problemes de concurrència degut a la mutabilitat de l'estructura. A més a més, en el moment que vulguem accedir al valor de l'àtom ho haurem de fer de la forma `@nom_atom`. D'aquesta forma obtindrem l'estructura a la que fa referència l'àtom, la qual és immutable i no hi ha risc de que canviï, i no el propi àtom. Cal destacar que les modificacions sobre un àtom són atòmiques, és a dir, mai podríem obtenir un valor parcialment canviat ni tindríem problemes en fer modificacions simultànies en ell.

```
(def atm (atom 1))          ; definim àtom amb valor 1
(println @atm)              ;imprimim el valor d'una copia de l'atom
```

Codi 2.1-7 Exemple àtom Clojure

2.1.2.5 Altres característiques

Una de les característica curioses alhora que útils és la possibilitat de tenir seqüències amb "avaluació mandrosa", la qual cosa és l'únic aspecte de "mandrositat" del llenguatge. Això significa que els elements de la seqüència no són avaluats fins que són necessaris, la qual cosa permet representar conjunts infinits. En el següent exemple podem veure com guardem una seqüència infinita representant els números naturals i agafem els cinc primers.

```
(def naturals (range))    ;seqüència de tots els números naturals
(take 5 naturals)  ;obtenim els 5 primers elements de la seqüència anterior
```

Codi 2.1-8 Exemple seqüència infinita Clojure

D'altra banda, donat que el llenguatge s'executa sobre la JVM, les aplicacions en Clojure poden ser fàcilment integrades en servidors d'aplicacions o altres entorns Java

amb escassa complexitat addicional. Així com també es poden utilitzar totes les llibreries de Java.

2.2 ClojureScript

En els últims anys les aplicacions web han passat a ser cada vegada més elaborades i amb contingut més "pessant", en quant a termes d'espai. Degut a això s'ha incrementat notablement el cost que suposa recarregar una pàgina web. A més a més, en moltes ocasions no seria necessari recarregar tota la pàgina degut a que només s'hi ha de fer un petit canvi.

És per això que cada vegada es tendeix més a fer el que s'anomena "aplicacions de navegador". Aquestes aplicacions segueixen una arquitectura client-servidor. El servidor s'encarrega de proporcionar les dades necessàries al client, responent a peticions d'aquest, i el client mostra aquestes dades. El client també té la funció de contactar amb el servidor per resoldre les interaccions que faci l'usuari. D'aquesta forma, el servidor no cal que enviï pàgines d'HTML, sinó que només enviï les dades. En un principi aquest tipus d'interacció només s'utilitzava per petits canvis que s'haguessin de realitzar en la web, conservant les pàgines principals, per les quals el servidor servia tot l'HTML. No obstant això, en els darrers temps es tendeix a fer clients molt complets i les pàgines web passen a ser el que s'anomena *"Single-page Application"*. És a dir, tenim una única pàgina (url) en la que el client s'encarrega de fer tota la interacció amb el servidor per servir la interacció de l'usuari.

El llenguatge que s'acostuma a utilitzar per implementar el client és JavaScript, el qual tot i tenir alguns defectes, és l'únic que tenim garantit que està implementat en tots els navegadors. No obstant això, per resoldre els defectes que presenta apareixen altres llenguatges que poden ser compilats a codi JavaScript per ser interpretat pel navegador. És el cas de ClojureScript [5].

2.2.1 Visió global del llenguatge

ClojureScript és una versió de Clojure que es pot compilar a JavaScript, per tant, es pot utilitzar en tots els llocs en què pot utilitzar-se JavaScript i també pot emprar les llibreries de JavaScript. D'altra banda, a part de tenir la mateixa sintaxi que Clojure, ClojureScript suporta també tota la semàntica [A1]. Incloent-hi les estructures de dades inmutables, seqüències mandroses, macros…

2.2.2 Compilació

ClojureScript és un compilador que transforma codi en llenguatge ClojureScript a codi JavaScript. Aquest està dissenyat per treballar conjuntament amb Google Closure Compiler, el qual optimitza el codi JavaScript per reduir-ne la mida. A més a més, disposa de varis tipus d'optimitzacións que permeten comprimir més o menys els arxius compilats. Amb la conseqüència que com més comprimits estan els arxius, més illegibles es fan per al desenvolupador. Els diferents tipus d'optimització són:

- **Només espais en blanc:** Aquesta opció només elimina els comentaris i espais innecessaris del codi JavaScript.
- **Optimitzacions simples:** Realitza les optimitzacións del tipus "Només espais en blanc" i renombra símbols i paràmetres de funcións per altres de més curts per tal de reduir la mida del codi JavaScript.
- **Optimitzacions avançades:** Realitza les optimitzacions dels dos modes anteriors i també renombra les funcions i les variables globals. A més, elimina el codi "mort" o que no s'utilitza.

Cal destacar que els tipus d'optimitzacions simples i avançades tenen algunes restriccions en quant al format que ha de tenir el codi per ser optimitzat satisfactòriament. Això suposa un problema en alguns casos, ja que les llibreries de JavaScript no acostumen a estar pensades per ser optimitzades amb Google Closure Compiler i, per tant, no compleixen amb les restriccions necessàries. No obstant això, podrem utilitzar igualment totes les llibreries de JavaScript amb l'única restricció de no

poder ser optimitzades amb Google Closure Compiler. En les últimes versions de ClojureScript, el qual ha canviat força al llarg del projecte, algunes d'aquestes restriccions queden simplificades. Tot i això, en el nostre projecte no es veuen reflectits els canvis en ClojureScript degut a que no hem actualitzat a les darreres versions d'aquest. El fet de no haver-ho fet ha estat degut a que volíem estabilitat en el projecte.

2.2.3 ReactJS

ReactJS [W6] és una llibreria de JavaScript desenvolupada per les companyies Facebook i Instagram conjuntament. Aquesta llibreria ajuda a construir eficientment interfícies d'usuari que mostren dades canviants en el temps. Per fer aquesta tasca, ens permet expressar com hauria de ser la nostra pàgina en qualsevol moment, independentment de les dades. Després, quan les dades canviïn s'encarregarà d'actualitzar-les automàticament.

La idea és crear l'estructura que presentarà la pàgina en qualsevol moment i deixar-la preparada per mostrar les dades que contingui una variable d'estat de l'aplicació. En el moment en què es modifiqui l'estat de l'aplicació es calcularan els canvis que s'han de realitzar en la pàgina i es produiran. Per tal de fer aquest procés eficientment, ReactJS crea un DOM virtual en representació del DOM real de l'aplicació i sobre el qual serà molt més eficient accedir. El DOM virtual és el que utilitza ReactJS per trobar els canvis que ha sofert respecte de l'última versió d'aquest. ReactJS utilitza algorismes per calcular els mínims canvis a realitzar sobre l'antic DOM per transformar-lo en l'actual. Una vegada calculada la llista de canvis a realitzar, aquests es produiran, ara si, sobre el DOM real de la pàgina web. D'aquesta manera resulta molt més eficient que si realitzéssim el mateix procés treballant tota l'estona sobre el DOM real.

D'altra banda, una possibilitat que ens ofereix ReactJS i que és important utilitzar per tal de fer més entenedor el codi, és la divisió de la nostra interfície d'usuari en components i subcomponents. Cada component representarà un fragment visual de la pàgina. A més, els components podran contenir altres components, fins a la unitat mínima.

En la següent imatge queda més clara aquesta idea.

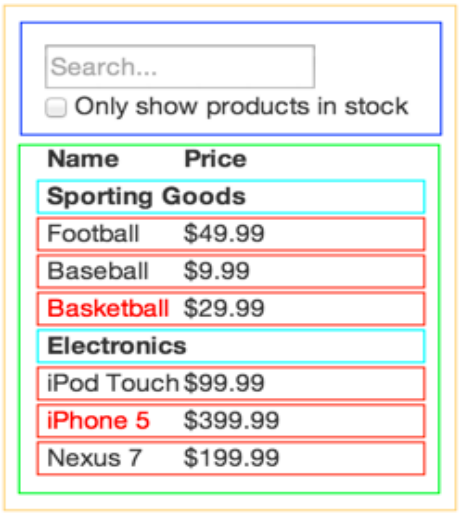


Figura 7 Representació de components Om

Els diferents quadres representen els diferents components. Podem veure que alguns components contenen altres components, fins arribar als components que representen un únic element de la interfície. Amb els diferents components d'aquest exemple podríem construir el següent arbre en el que es mostra com els diferents components estan dins de components més grans.

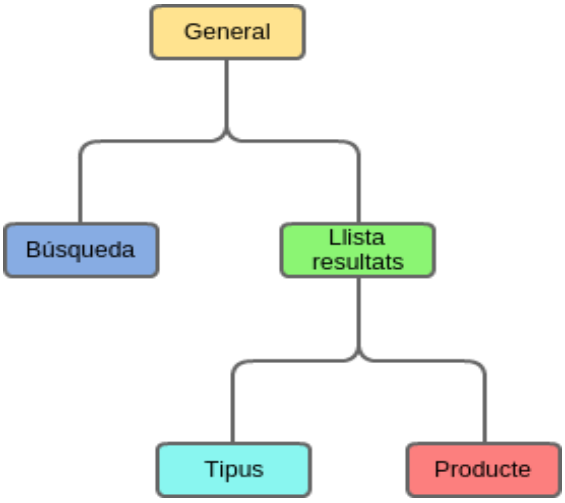


Figura 8 Esquema components Om

A més a més, una altra peculiaritat de ReactJS que lliga amb els components és el fet de poder desglossar l'estat de l'aplicació en els diferents components. És a dir, en l'exemple anterior el component que fa referència als productes només necessita veure la part de l'estat que conté els productes.

2.2.4 Om

En el nostre projecte no utilitzarem ReactJS directament, ja que aquest està pensat per utilitzar-se des de JavaScript i nosaltres estem treballant en ClojureScript, tot i que, com ja hem comentat, podem utilitzar les llibreries de JavaScript però resulta una mica feixuc. A canvi, utilitzarem Om [R9], el qual fa d'adaptador entre ClojureScript i ReactJS i ens permet fer un ús més còmode d'aquesta llibreria [A7]. A més a més, Om canvia una mica alguns aspectes respecte a ReactJS. Això es deu a que per les característiques de ClojureScript, alguns aspectes de ReactJS poden millorar-se en quant a rendiment o estructuració del codi.

Om és una llibreria de ClojureScript que utilitza per sota ReactJS i ofereix pràcticament les mateixes característiques. A més a més, algunes de les característiques de ClojureScript fan que Om funcioni encara millor que ReactJS en alguns aspectes. El més important d'aquests aspectes és la millora de rendiment que suposa fer la comparació entre els DOMS virtuals de ReactJS amb els elements immutables de ClojureScript.

Donat els arbres que els estats dels components (actual i modificat) d'una pàgina web, ReactJS compararia cadascun dels nodes dels arbres per trobar les diferències entre ells i veure quins components cal reconstruir-los. En canvi, en el cas d'Om s'hauria de fer el mateix procés però amb la peculiaritat que, donat que les estructures de dades són persistents, no caldria recórrer tots els nodes de l'arbre si només hi ha canvis en alguns dels nodes. Només seria necessari continuar comparant els nodes descendents de nodes que no siguin la mateixa instància en els dos estats, ja que en ser la mateixa instància, no ha pogut canviar res en la resta d'arbre descendent.

Tenint present com es representen les estructures de dades en Clojure, la qual cosa hem explicat en l'apartat **Estructures de dades immutables i persistents**, podem imaginar com és la comparativa que fa Om, ja que, com hem vist, en modificar una estructura de dades molts dels seus nodes són compartits en l'estructura nova i la vella.

D'altra banda, en Om tindrem l'estat de l'aplicació en un àtom com podria ser el següent:

```
(def app-state
 (atom
   {:people
    [{:type :student :first "Ben" :last "Bitdiddle" :email "benb@mit.edu"}
     {:type  :student :first "Alyssa" :middle-initial "P"
      :last "Hacker" :email "aphacker@mit.edu"}
     {:type  :professor :first "Gerald" :middle "Jay" :last "Sussman"
      :email "metacirc@mit.edu" :classes [:6001 :6946]}
     {:type :student :first "Eva" :middle "Lu" :last "Ator"
      :email "eval@mit.edu"}
     {:type :student :first "Louis" :last "Reasoner"
      :email "prolog@mit.edu"}
     {:type    :professor :first "Hal" :last "Abelson"
      :email "evalapply@mit.edu" :classes [:6001]}]
    :classes
    {:6001 "The Structure and Interpretation of Computer Programs"
     :6946 "The Structure and Interpretation of Classical Mechanics"
     :1806 "Linear Algebra"}}))
```

Codi 2.2-1 Exemple estat aplicació Om

Sobre aquest estat Om crearà la pàgina web i, quan aquest canviï, la modificarà. No obstant això, una característica que diferència Om respecte a ReactJS és que els diferents components sovint no treballen sobre la totalitat de l'estat, sinó que només tenen accés a una part d'aquest. Per exemple, en l'exemple anterior podríem tenir un component que només tingués accés a la part `:people`, ja que és l'encarregat de crear la part de la pàgina web corresponent a aquesta part. No obstant això, si aquest

component modifica la seva part de l'estat, els canvis si que es veuran reflectits en l'estat global de l'aplicació. A més a més, d'aquesta forma podem crear components reutilitzables que no depenguin de l'estat de l'aplicació, sinó només d'un subconjunt d'aquest.

2.3 Descripció aplicacions web Clojure/ClojureScript

2.3.1 Descripció

L'estructura d'una aplicació web en Clojure [6] no canvia gaire respecte al que podem veure en altres entorns, ja que en un entorn web les idees són les mateixes independentment del llenguatge que estem utilitzant. No obstant això, hi ha algunes coses que criden l'atenció. Si volguéssim desenvolupar una aplicació web en algun altre llenguatge segurament utilitzaríem un *framework* per evitar fer codi repetitiu i feixuc. Això comporta haver d'aprendre el funcionament intern d'aquest, ja que en alguns casos hi ha coses que poden semblar "màgiques" en la major part dels casos resulta difícil entendre el codi sense conèixer a fons el *framework*. En Clojure no disposarem d'un *framework* sinó de diverses llibreries més específiques per a cadascuna de les funcionalitats que vulguem cobrir. D'aquesta forma aconseguirem tenir un codi una mica més entenedor que quan s'utilitza un *framework*, ja que en aquest cas es veu força clar d'on prové cada cosa.

2.3.2 Servidor

Com en qualsevol altre llenguatge, en la part del servidor tindrem les url sobre les quals es podrà rebre peticions associant a cada petició el codi que donarà una resposta. En el procés de generar una resposta és possible que es produeixin accions contra una base de dades per guardar o recuperar informació necessària per a la resposta. A més, un fet sorprenent i poc habitual en altres llenguatges és la possibilitat de generar el codi

html de la resposta directament en el codi Clojure, sense necessitat de crear una pàgina d'html.

La definició de les url en Clojure es fa mitjançant la funció `defroutes`, tal i com podem veure en el següent fragment de codi.

```
(defroutes auth-routes
  (GET "/register" [_] (registration-page))
  (POST "/register" [id pass pass1]
    (handle-registration id pass pass1))
  (GET "/login" [] (login-page))
  (POST "/login" [id pass]
    (handle-login id pass))
  (GET "/logout" []
    (layout/common
      (form-to [:post "/logout"]
         (submit-button "logout"))))
 (POST "/logout" []
    (session/clear!)
    (redirect "/")))
```

Codi 2.3-1 Exemple definició rutes Clojure

Podem observar que per a cada url indiquem, a banda del mètode amb el que es sol·licita la url, els paràmetres que rebrem per a poder donar una resposta degut a un formulari o alguna cosa semblant. També s'indica la funció que es cridarà per generar la resposta passant-li els paràmetres que hem rebut com a paràmetres. A més a més, no és necessari definir totes les rutes de l'aplicació web en un mateix arxiu, podem definir-les en cada arxiu només les rutes a les quals dóna resposta. D'aquesta forma podrem obtenir un codi més estructurat. Posteriorment, el que si que serà necessari és ajuntar les diferents definicions de rutes, la qual cosa es farà gràcies a la funció `site` de la llibreria Compojure. Cal destacar que un dels avantatges de la sintaxi de Lisp el que permet crear mini-llenguatges o DSLs (*Domain Specific Languages*) que permeten,

com en aquest cas, adaptar la sintaxi del llenguatge a l'especificació de rutes d'un servidor web.

```
(def app
  (->
    (handler/site
      (routes category-routes
              pilot-routes
              auth-routes
              home-routes
              app-routes))
    (wrap-base-url)
    (session/wrap-noir-session
      {:store (memory-store)})
```

Codi 2.3-2 Exemple definició rutes Compojure

Aquest símbol s'utilitzarà cada vegada que es posi en funcionament el servidor i conté tot el necessari per al funcionament del servidor. També ens serveix per activar algunes funcionalitats del servidor que proporcionen certes llibreries. En l'exemple anterior, activem l'ús de sessions i la validació de formularis mitjançant les funcions `wrap-noir-session` i `wrap-noir-validation` de la llibreria `noir`.

Finalment, sorprèn el fet de poder generar codi html des del propi codi Clojure. Això ens ho permet fer la llibreria hiccup i podem fer coses com les següents:

```
(defn common [& body]
  (html5
    [:head
      [:title "Welcome to motogp"]
      (include-CSS "/CSS/base.CSS")]
      [:body body]))
```

Codi 2.3-3 Exemple codi html amb hiccup

En aquesta funció estem creant codi en html5 en el codi Clojure. Podem observar que el codi es representa mitjançant vectors. A més a més, veiem que estem creant una funció que rep un paràmetre i després el posem en el "body" del html. Amb això aconseguim crear un patró en el qual només ens cal passar-li el "body" de l'html en format de vectors com el que utilitza hiccup i ens crearà la pàgina sobre la plantilla que hem fet. D'altra banda, en cas que vulguem donar valor a algun dels atributs dels elements d'html es fa mitjançant una taula associativa, de la següent forma:

```
[:a {:href "http://GitHub.com"} "GitHub"]
```

Codi 2.3-4 Exemple propietats html amb hiccup

En programació funcional és força habitual fer funcions que generen estructures de dades que després s'interpreten. En el nostre cas, generem estructures de dades que representen les pàgines que després s'interpreten i generen la pàgina. La idea és que les estructures de dades són força més composables que els productes finals.

2.3.3 Client

El client no és imprescindible per una aplicació web en Clojure, igual que tampoc ho és per a les realitzades en altres llenguatges. No obstant això, podem delegar-li tota la part de crear la interfície d'usuari. De fet aquesta és la forma que utilitzarem en el nostre projecte mitjançant Om, ja explicat anteriorment.

2.3.4 Core.async

En la major part dels casos, quan s'executen varis processos concurrentment en un programa, es necessita comunicació entre els diferents processos per passar-se certa informació [1]. La biblioteca "core.async" permet fer aquesta comunicació mitjançant canals.

Un canal permet transmetre informació entre diferents processos de forma bidireccional, és a dir, qualsevol tasca amb una referència a un canal por enviar i rebre informació per ell. A més a més, ni qui envia té necessitat de saber qui està escoltant el canal, ni viceversa.

No obstant, un dels problemes que presenten aquests canals és que en el moment en que una tasca envia alguna cosa per un canal, es queda bloquejada esperant que algú llegeix-hi el canal. El mateix passa quan una tasca espera rebre d'un canal, es queda bloquejada fins a rebre alguna cosa. Això suposa un problema en alguns casos, ja que moltes vegades el programa podria continuar la seva execució sense problemes tot i no esperar a que es completi la comunicació pel canal. En el següent exemple mai s'arribarà a imprimir el test "Hola" degut a que l'execució es quedarà parada en la segona línia esperant que algú rebi el que s'ha enviat pel canal.

```
(def canal (chan))          ; definim canal
(>!! canal "Hola!")         ; enviem "Hola" pel canal
(println (<!! canal))       ; llegim del canal
```

Codi 2.3-5 Exemple canal bloquejat

2.3.4.1 Go Blocks

Els Go Blocks són blocs de codi que s'executen de forma asíncrona. D'aquesta forma evitem el problema de quedar-se les tasques bloquejades al llegir o escriure en un canal, ja que el codi que no estigui dins del bloc Go es continuarà executant de forma independent. A més a més, en llegir i escriure de canals dins d'un bloc Go podem fer-ho de manera que en el moment en que un fil es quedi bloquejat en un canal, aquest fil quedi "aparcat" i no consumeixi memòria fins que es completi la operació de lectura o escriptura en el canal. En la següent versió de l'exemple anterior utilitzem Go Blocks per tal de que no es quedi parada l'execució, ara si s'imprimirà el text.

```
(def canal (chan))
(go (>! canal "Hola!"))
(go (println (<!! canal)))
```

Codi 2.3-6 Exemple canal amb blocs Go

2.3.5 Websocket

Un tret característic de l'arquitectura client-servidor en Clojure és la possibilitat d'utilitzar *websocket* per connectar client i servidor, tot i que en altres llenguatges també és possible utilitzar aquest mètode. Un *websocket* és un canal de comunicació bidireccional (semblant als que hem vist en l'apartat anterior) que s'estableix entre el client i el servidor i els permet comunicar-se sense necessitat que el client enviï peticions GET o POST al servidor.

Per establir un *websocket* entre client i servidor cal que el servidor estigui configurat correctament per poder acceptar peticions dels clients per obrir un *websocket*. Cal destacar que sempre serà el client l'encarregat de fer el primer pas per obrir el *websocket*, enviant una petició al servidor. En el moment en què aquest accepta la petició, dóna la confirmació al client i s'estableix la connexió. A partir d'aquest moment poden comunicar-se en temps real pel canal.

En Clojure, la llibreria sente [R9] ens permet utilitzar *websockets* de la següent forma:

- **Servidor**

```
(let [{:keys [ch-recv send-fn ajax-post-fn
              ajax-get-or-ws-handshake-fn connected-uids]}
      (sente/make-channel-socket! {})]
 (def ring-ajax-post ajax-post-fn)
 (def ring-ajax-get-or-ws-handshake ajax-get-or-ws-handshake-fn)
 (def ch-chsk ch-recv)                ; canal de rebuda
 (def chsk-send! send-fn)             ; canal d'enviada
 (def connected-uids connected-uids)) ; UID dels clients connectats
```

Codi 2.3-7 Exemple part servidor websocket

Podem veure que disposem de tot el necessari per comunicar-nos amb els diferents clients. Així doncs, només caldrà que es quedi escoltant el canal de rebuda i doni resposta a les peticions, o directament enviï informació als clients. A més a més, els clients que es van connectant al *websocket* queden guardats en un àtom mitjançant el seu `uid`, el qual fa d'identificador del client. Després, cada vegada que vulguem enviar alguna cosa haurem de dir a quin client volem enviar-ho.

D'altra banda, caldrà que definim les rutes sobre les quals els clients hauran d'enviar la petició per establir una connexió via *websocket*. Ho farem de la manera habitual mitjançant la funció `defroutes` i de la següent forma:

```
(GET "/chsk" req (ring-ajax-get-or-ws-handshake req))
(POST "/chsk" req (ring-ajax-post req))
```

Codi 2.3-8 Rutes per websocket

- **Client**

```
(let [{:keys [chsk ch-recv send-fn state]}
      (sente/make-channel-socket! "/chsk" ; Ruta definida al servidor
                                  {:type :auto })]
 (def chsk chsk)
 (def ch-chsk ch-recv)                    ; Canal per rebre
 (def chsk-send! send-fn)                 ; Funció per enviar
 (def chsk-state state))                  ; Estat
```

Codi 2.3-9 Exemple part client

Com en el cas del servidor, en crear un *websocket* la funció ens retorna una taula associativa amb tot el necessari per fer la comunicació amb el servidor. Cal destacar que hem d'indicar la ruta sobre la qual enviem la petició d'obrir el *websocket*, la qual haurà d'estar definida i configurada correctament en el servidor per establir la connexió.

2.4 Infraestructura i eines

2.4.1 Git

Git s'ha convertit en els últims anys en una de les eines de control de versions de software més utilitzades, sobretot en projectes nous i de software lliure. Va ser dissenyat per Linus Torvalds pensant en l'eficiència i confiabilitat del manteniment de versions en un entorn amb un gran nombre d'arxius de codi font. Nosaltres utilitzarem Git per fer el control de versions del nostre projecte, la qual cosa ens permetrà tenir versions estables del projecte en tot moment i poder recuperar una versió estable en cas que arribem a un punt d'inconsistència del codi i no puguem arreglar-ho.

En el nostre projecte crearem dues branques per tal de tenir per separat el codi que està en desenvolupament de les versions estables. Així doncs, tindrem una branca sobre la qual guardarem els avanços que anem fent en les diferents iteracions, sense necessitat d'haver acabat la iteració. D'altra banda, en la branca "master" tindrem les versions estables del projecte, les quals es correspondran amb les diferents iteracions d'aquest. D'aquesta forma sempre disposarem de la versió estable de l'última iteració desenvolupada.

2.4.2 GitHub

GitHub és una web de *hosting* de projectes amb control de versions Git. Ens permet pujar el nostre codi per a poder veure'l, descarregar-lo o pujar una nova versió des de qualsevol lloc. Ofereix totes les funcionalitats de Git a més d'algunes pròpies com la visualització gràfica dels projectes en la pàgina web.

Nosaltres utilitzarem aquesta eina per tal de tenir disponible el codi des de qualsevol lloc i poder treballar des de diferents llocs. A més a més, també ens serà molt útil per tal poder compartir el projecte fàcilment amb el tutor del projecte i comentar algun aspecte o resoldre algun dubte. Fins i tot GitHub dóna opció a que persones no propietàries d'un projecte facin el que s'anomena una "*Pull request*", la qual cosa seria

com demanar al propietari del projecte incorporar en el seu projecte canvis realitzats per aquesta persona. Això ens serà útil en cas que el tutor ens tingui que ajudar a causa d'algun problema en el codi. Amb aquesta eina podrem incorporar fàcilment canvis que hagi realitzat per resoldre'ns el problema.

2.4.3 IntelliJ

IntelliJ IDEA és un entorn integrat de desenvolupament Java (IDE) creat per JetBrains i està disponible de forma gratuïta (Edició Comunitària) o de pagament (Edició Comercial). La primera versió va sortir el gener de 2001, essent un dels primers Java IDE disponibles amb navegació de codi avançada i eines per la refactorització de codi.

Per al nostre treball, en tindrem suficient amb la versió comunitària (gratuïta), ja que aquesta permet fer el desenvolupament en Clojure gràcies al *plugin* Cursive [W4], que l'explicarem posteriorment. Cal destacar que sense aquest *plugin* IntelliJ no està preparat per desenvolupar projectes en Clojure.

D'altra banda, aquest entorn de desenvolupament ens serà molt útil per al desenvolupament del nostre projecte, ja que ens permetrà tenir unificades les eines necessàries per a desenvolupar i posar en funcionament l'aplicació web. Per tant, l'únic que necessitarem, a banda de l'IDE, per al desenvolupament del projecte serà un navegador on poder veure l'aplicació web. En canvi, si no disposéssim d'una eina com InteliJ i del *plugin* Cursive, seria necessari un terminal per compilar i executar la nostra aplicació.

D'altra banda, com la resta d'entorns de desenvolupament, aquest també disposa d'eines per treballar amb Git i GitHub, la qual cosa també ens serà útil per fer el control de versions del nostre projecte sense haver d'utilitzar les sentències de Git pel terminal, la qual cosa resulta més feixuga.

Finalment, cal dir que després d'haver desenvolupat una mica en aquest entorn m'ha semblat de molta utilitat degut a les facilitats que dóna per al desenvolupament del codi. A més a més, ofereix un sistema per controlar la correcta estructura dels parèntesis del

codi. Al principi sembla una mica robust, ja que és molt restrictiu i impedeix que hi hagi un parèntesis obert sense estar tancat en qualsevol moment. No obstant això, al final t'adones que això és molt útil a mesura que van incrementant les línies de codi per tal de tenir sempre el codi ben estructurat en tot moment. I més encara en Clojure, on tot es basa en parèntesis i no s'utilitza un sistema d'indentació que permeti veure a simple vista on comencen i acaben els parèntesis.

2.4.4 Cursive

Cursive [W4] és un *plugin* d'IntelliJ que permet utilitzar l'IDE per a desenvolupar aplicacions en Clojure, ja que el propi IntelliJ no incorpora les característiques per al desenvolupament en Clojure sense aquest *plugin* i es comportaria com un editor de text normal.

Gràcies a aquest *plugin*, IntelliJ es comporta com ho fa en altres llenguatges per als que si estan preparats, proporcionant les següents característiques:

- **Ressaltat del codi:** es ressalta el codi per fer-lo més entenedor a simple vista.
- **Navegació:** eines de navegació pel codi, per veure usos d'una funció, anar a la implementació d'una funció…
- **Renombrar símbols:** permet renombrar fàcilment funcions o símbols, renombrant automàticament els usos d'aquests.
- **REPL:** incorpora un `REPL` a l'entorn de desenvolupament. En l'apartat **El REPL** explicarem què és un `REPL`.
- **Suport per Leiningen:** incorpora a l'entorn funcionalitats per treballar amb Leiningen.
- **Editat estructurat:** és restrictiu amb l'estructura del codi i no permet que hi hagi codi desestructurat a causa de parèntesis incoherents.
- **Formatejat del codi:** reestructura automàticament el codi per deixar-lo més llegible.
- **Funcions de debug.**

- **Integració amb Java per a projectes mixtes.**

2.4.5 Leiningen

Leiningen és una eina que automatitza el muntatge i el control de dependències d'un projecte en Clojure. Les característiques que presenta són:

- Creació de l'estructura d'un nou projecte a partir de plantilles.
- Resolució de dependències amb descarrega automàtica de les llibreries.
- Iniciar un `REPL` interactiu amb les dependències del projecte carregades per poder executar-lo.
- Empaquetar el codi del projecte i les dependències en un arxiu ".jar".

Al igual que altres eines que realitzen la mateixa feina en altres llenguatges, per utilitzar Leiningen cal crear un arxiu de configuració, en aquest cas anomenat "project.clj". En ell definírem les dependències del projecte, les quals haurem de definir en el format correcte i indicant la versió que volem utilitzar. Amb això, Leiningen descarrega automàticament les llibreries necessàries per poder ser utilitzades en el projecte. A més a més, també podem definir en aquest mateix arxiu informació del projecte (descripció, `url` de la pàgina del projecte, llicència…), paràmetres per a compilar i muntar el projecte, *plugin*s de Leiningen i paràmetres per a la utilització d'aquests…

2.4.6 El REPL

El `REPL` és present en els llenguatges Lisp, que tot i ser llenguatges compilats permeten treballar com es podria fer en altres llenguatges interpretats. Es tracta d'una mena de consola en la que podem introduir expressions de codi que seran avaluades i se'ns retornarà un resultat. D'aquí provenen les sigles `REPL`, les quals signifiquen "*Read-Eval-Print-Loop*" (Llegir-Avaluar-Imprimir-Repetir). És a dir, el seu funcionament és:

1. Llegir un input.
2. Avaluar-lo.

3. Imprimir el resultat.
4. Repetir el procés.

Així doncs, mitjançant el `REPL` podem experimentar amb el llenguatge i provar els nostres programes de forma interactiva, la qual cosa és molt útil donada la rapidesa que suposa aquest mètode respecte a haver de compilar el programa per provar alguna cosa. D'altra banda, també ens permet carregar codi d'algun fitxer i cridar les funcions que hi hagi en aquest. Cosa que també és molt beneficiosa per tal de debugar troços de codi.

2.4.7 Figwheel

Figwheel és una llibreria de ClojureScript molt útil per desenvolupar codi en aquest llenguatge. Aquesta llibreria permet visualitzar els canvis que fem en el codi ClojureScript automàticament en el navegador. Amb la qual cosa no és necessari parar el servidor i tornar a posar-lo en funcionament per veure els efectes dels canvis. A més a més, la llibreria també ens permet veure automàticament els canvis que realitzem en el CSS de la pàgina web. Cal destacar que el temps que triga en recarregar el codi modificat és mínim degut a que només recarrega els canvis i no tot el codi.

D'altra banda, Figwheel permet iniciar un `REPL` de ClojureScript connectat directament al codi que s'està executant en el nostre navegador. Amb la qual cosa podem fer proves o executar trossos de codi manualment i tot es veurà reflectit en el navegador.

Per poder utilitzar Figwheel cal configurar la nostra aplicació per treballar amb les funcionalitats d'aquesta llibreria. En aquest sentit no hem sigut nosaltres els encarregats de codificar aquesta configuració, ja que hem utilitzat la plantilla "chestnut" per crear el projecte, la qual ja incorpora aquesta configuració.

2.4.8 Google Docs

Cal destacar que durant tot el treball hem utilitzat Google Docs com a eina de processament de text, ja que d'aquesta forma podíem tenir el document de la memòria compartit en tot moment amb el tutor del treball i comentar amb facilitat els canvis en aquesta. No obstant això, per tal de realitzar la versió final del projecte hem utilitzat Microsoft Word, ja que Google Docs encara té algunes mancances en quan a estils.

3 ANÀLISI DE REQUERIMENTS

3.1 Requeriments inicials

En aquest apartat definirem els requeriments inicials per tenir una idea en tot moment de les tasques a realitzar i realitzades. No obstant això, és probable que haguem de refinar els requeriments inicials abans d'implementar-los per tal d'adaptar-los a les nostres possibilitats temporals i funcionals. A més a més, com ja hem mencionat anteriorment, l'objectiu del projecte no és cenyir-nos a una idea i a uns requeriments sinó l'aprenentatge i ús d'una combinació de tecnologies, usant una arquitectura determinada d'aplicació. Per tant, els requeriments que definim en aquest apartat només són una "excusa" per a practicar les tecnologies que hem aprés.

En el nostre cas la recerca dels requeriments no és una tasca difícil degut a que no disposem d'un client que ens demana una certa aplicació, sinó que som nosaltres mateixos els que hem decidit quines funcionalitats tindrà la nostra aplicació i quines no. En canvi, el que si que haurem de valorar per definir els requeriments és el temps del que disposem per al desenvolupament de l'aplicació, ja que no seria lògic definir uns requeriments molt complexos o molt extensos sabent que no disposarem de temps suficient per assolir-los. A més a més, també haurem de tenir en compte la complexitat que suposarà implementar certes funcionalitats. Si definim funcionalitats molt complexes segurament utilitzarem molt temps en elles i els avanços que quedaran reflectits en l'aplicació no seran gaire notoris.

Cal destacar que estem treballant en un àmbit que en un principi era totalment nou per a nosaltres i ha sigut necessari un aprenentatge previ per poder desenvolupar l'aplicació. No obstant això, la nostra experiència continua sent escassa i estem desenvolupant la nostra primera aplicació web en Clojure. Per tant, en aquest projecte no estem buscant fer una cosa molt complexa ni útil, sinó que definim els requeriments del projecte intentant explorar i practicar amb el llenguatge i les llibreries d'aquest.

Partint de les bases mencionades anteriorment, els requeriments que inicialment definim per a la nostra aplicació web són els següents:

El sistema ha de connectar-se amb l'API de Twitter per tal de rebre l'*stream* de *tweets*.

El sistema ha de mostrar els *tweets* que li arribin en una pàgina web en temps real.

El sistema ha de calcular i guardar la longitud dels *tweets* per tal de mostrar una estadística d'aquests.

El sistema ha de mostrar en la pàgina web una gràfica en la que es representi l'estadística de la longitud dels *tweets*.

El sistema ha de calcular i guardar el país de procedència dels *tweets* per tal de mostrar-ho estadísticament.

El sistema ha de mostrar en la pàgina web un mapa mundial en la que es representaran els llocs d'on estem rebent *tweets*, també quedarà diferenciada la proporció de *tweets* que rebem de cada lloc.

El sistema permetrà seleccionar una de les franges de longitud dels *tweets* i mostrarà, en aquest cas, només la informació del país d'on provenen els *tweets* d'aquesta longitud.

El sistema permetrà seleccionar un país i mostrarà, en aquest cas, només la informació de la longitud dels *tweets* que provenen d'aquest país.

Cal destacar que som conscients de la complexitat que suposen els dos últims requeriments i és molt probable que no disposem de temps suficient per arribar a implementar aquesta part. Tot i això, els definim perquè no podem fer una estimació prou acurada del temps que suposarà implementar i documentar la resta de requeriments. D'aquesta forma, en funció del temps que resti fins a la presentació del treball decidirem si implementar o no aquestes funcionalitats. No obstant això, en cas de no definir aquestes dos funcionalitats podríem trobar-nos que haguéssim acabat d'implementar totes les funcionalitats i ens sobrés molt de temps fins a la presentació del treball.

3.2 Planificació inicial: iteracions

Donats els requeriments de l'apartat anterior, definirem unes iteracions per tal d'anar desenvolupant-los progressivament. Aquest sistema també ens ajudarà a planificar la feina i deixar versions estables de l'aplicació conforme anem superant les diferents iteracions. Habitualment, les iteracions es correspondran amb un o dos dels requeriments definits inicialment. D'altra banda, en cada iteració també redactarem la part de la memòria escrita corresponent a ella.

La primera iteració es correspon amb els requeriments 1 i 2. Aquesta iteració l'hem fet com una feina d'exploració del llenguatge, de la llibreria Om i de la API de Twitter, fins que, gairebé sense voler-ho, ens hem trobat amb una primera versió del projecte que documentarem com a primera iteració.

La segona iteració anirà relacionada amb el requeriment 3, en ella definirem uns intervals en els quals classificarem els *tweets* segons la seva longitud a mesura que ens vagin arribant. Cal destacar que, no guardarem tota la informació del *tweet* sinó que només incrementarem el comptador de l'interval. D'aquesta forma acabarem obtenint el nombre de *tweets* que ens han arribat de cada interval. D'altra banda, en aquesta iteració ho mostrarem en la pàgina web en forma de text pla, ja que la tasca de mostrar-ho en un gràfic afegeix força complexitat degut a que haurem d'utilitzar llibreries específiques per realitzar aquesta tasca, la qual cosa comporta haver d'aprendre a utilitzar-les. És per això que hem decidit deixar aquest apartat per a la següent iteració.

La tercera iteració es correspon amb el requeriment número 4 i va lligat amb el requeriment anterior, en el qual haurem calculat les estadístiques de la longitud dels *tweets*. En aquesta iteració ens encarregarem de mostrar les dades obtingudes en la iteració anterior de forma gràfica. Com ja hem esmentat anteriorment, per fer-ho haurem d'utilitzar una llibreria de ClojureScript que actualment desconeixem i, per tant, haurem d'aprendre a utilitzar-la.

Les iteracions quatre i cinc seran molt semblants a les dues anteriors, ja que també es tracta de calcular una estadística i mostrar-la gràficament. En aquest cas, tal com s'indica en els requeriments 5 i 6, la dada estadística que calcularem serà el país d'on prové el *tweet*. A més a més, amb una mica de sort podrem utilitzar la mateixa llibreria que haurem utilitzat per mostrar la gràfica en la iteració 3. No obstant això, en la iteració 4 només mostrarem les dades en text pla i serà en la cinquena iteració quan ho mostrarem mitjançant un mapa mundial.

En aquest punt ja haurem obtingut un treball força complert i podríem dir que ja hauríem assolit els objectius fonamentals del treball. No obstant això, en funció del temps del que disposem fins a l'entrega del treball decidirem si continuem per intentar satisfer algun dels dos requeriments restants, la qual cosa suposaria una altra iteració per a cadascun d'ells. A més, donat que són força independents entre ells, caldria decidir per quin dels dos comencem valorant la complexitat d'aquests.

3.3 Planificació temporal de les iteracions

En el següent diagrama de Gantt es mostra el que hem fet fins ara, el punt on ens trobem i la planificació temporal prevista per a la resta de projecte.

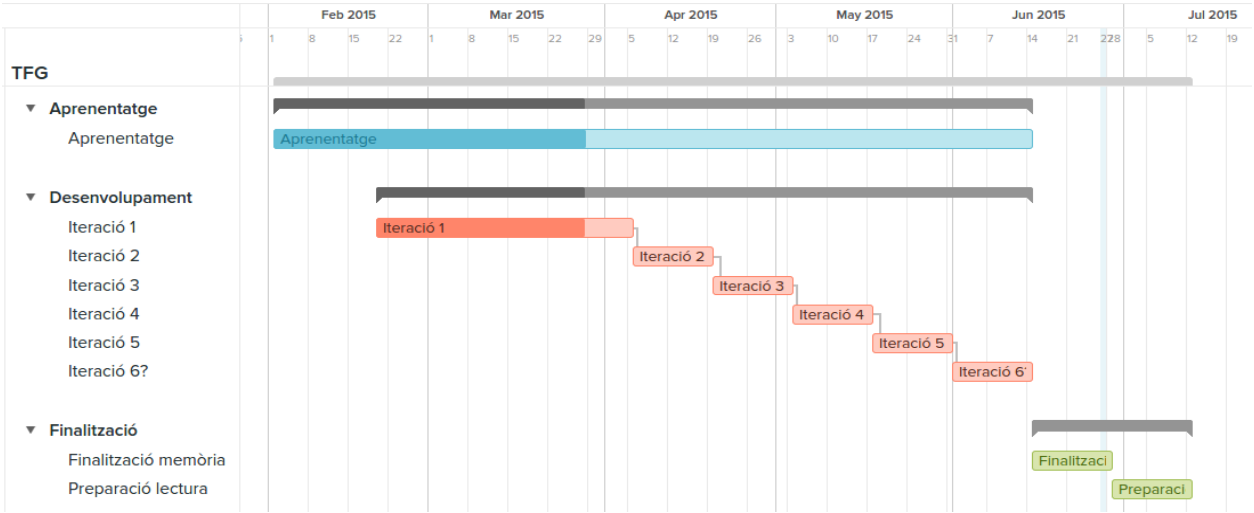


Figura 9 Diagrama de Gantt de planificació temporal de les iteracions

Podem observar que la tasca d'aprenentatge marca l'inici del projecte i acaba en el mateix moment que la última iteració d'aquest. Això és perquè, tot i que inicialment tenim un període en el qual ens vam centrar exclusivament a aprendre les tecnologies que anàvem a utilitzar, considerem que durant el transcurs de les iteracions també estarem aprenent les diferents tecnologies que anem utilitzant. De fet, segurament serà el moment en que més aprendrem, ja que amb la pràctica acabarem d'entendre les idees de les quals ens hem estat documentant.

D'altra banda, podem veure que ens trobem en la part final de la primera iteració, la qual té una durada molt llarga. No obstant això, com ja hem comentat aquesta iteració l'hem anat fent per provar algunes de les coses que anàvem aprenent i a partir del projecte "Om-twitter" del Juan Manuel Gimeno [R1], amb el qual hem anat "jugant" fins a arribar a tenir gairebé satisfets els requeriments de la primera iteració. Tot i això, en el diagrama veiem que hem planificat una setmana més per a aquesta iteració, en la qual haurem d'acabar de polir alguns aspectes i documentar-la.

Respecte a les següents iteracions, podem observar que hem planificat cada iteració en dues setmanes, ja que contem que durant la primera implementarem el codi de la iteració i en la segona documentarem la part corresponent a aquesta en la memòria escrita.

Finalment, hem previst aproximadament dues setmanes després de les iteracions per acabar de polir la memòria i deixar-la enllestida per a presentar-la. També hem previst dues setmanes més per tal de preparar la lectura del treball. Cal destacar que potser és una mica excessiu un mes per acabar la memòria i preparar la lectura. No obstant això, en cas d'endarrerir-nos en alguna de les iteracions ens anirà bé tenir una mica de marge.

4 DESENVOLUPAMENT

4.1 Iteració 1

4.1.1 Objectius

L'objectiu principal d'aquesta primera iteració és fer una primera versió del projecte que ens serveix-hi de base per al desenvolupament de la resta de funcionalitats del projecte. Així doncs, el primer que haurem de fer serà connectar el servidor de l'aplicació amb l'API de Twitter per rebre l'*stream* de *tweets* d'aquesta.

Una vegada assolit aquest primer objectiu, entrarà en joc el client de l'aplicació. Aquest s'encarregarà de mostrar els *tweets* que vagin arribant al servidor. Per poder fer aquest procés serà necessari connectar el servidor i el client per tal que el servidor pugui enviar els *tweets* al client i aquest pugui mostrar-los. Aquesta connexió la farem mitjançant un *websocket*.

4.1.2 Refinament dels requeriments

En primer lloc, un paràmetre que es pot indicar en iniciar la connexió amb l'API de Twitter, és un conjunt de paraules que s'utilitzaran per filtrar els *tweets* que ens arribin. Així doncs, només ens arribaran els *tweets* que continguin alguna de les paraules indicades. En el nostre cas, hem decidit posar un conjunt de paraules relacionades amb l'àmbit en el que estem treballant per tal d'evitar, en la mesura del possible, que ens arribin *tweets* en idiomes amb diferent alfabet al qual estem acostumats i que, per tant, no entendrem.

D'altra banda, el segon requeriment diu:

> *El sistema ha de mostrar els tweets que li arribin en una pàgina web en temps real.*

En aquesta iteració refinarem aquest requeriment, ja que només mostrarem els deu últims *tweets* rebuts. En cas contrari, s'acumularien en la pàgina web i no quedaria lloc per a les altres funcions de la pàgina. A més a més, l'objectiu no és veure tots els *tweets* sinó implementar la funcionalitat de mostrar-los, així que amb deu en tenim suficient.

4.1.3 Disseny i implementació del servidor

Anteriorment ja hem explicat com és un servidor en Clojure (**Servidor**) i tot el necessari per posar-lo en funcionament. L'estructura bàsica del projecte s'ha creat automàticament mitjançant la plantilla "chestnut". A partir d'aquí, el primer que hem fet ha sigut connectar el servidor a l'API de Twitter. Per fer-ho ens hem hagut de registrar a l'API per tal d'obtenir els nostres credencials, necessaris per utilitzar l'API. Posteriorment, hem hagut de crear un arxiu amb la configuració i els credencials necessaris per connectar-nos a l'API. Aquest arxiu serà utilitzat en el moment d'iniciar la connexió, la qual cosa fem de la següent forma:

```
(defn start-Twitter-conn! [conn chunk-chan]
  (log/info "Starting Twitter client.")
  (reset! conn (tas/statuses-filter
                :params {:track (:track conf)}
                :oauth-creds (creds conf)
                :callbacks (tweet-chunk-callback chunk-chan))))
```

Codi 4.1-1 Iteració 1: funció start-Twitter-conn!

Aquesta funció rep dos paràmetres, el primer és un àtom que conté la informació de la connexió amb l'API de Twitter, inicialment no contindrà res. El segon és el canal pel qual s'enviaran els *tweets* tal com vagin arribant a l'aplicació i en l'altre extrem d'aquest canal es processaran.

Cal destacar que en alguns casos l'API de Twitter envia alguns dels *tweets* en dues o més parts, amb la qual cosa serà necessari ajuntar les diferents parts per obtenir els

tweets d'una sola peça. Per fer aquest procés ho fem mitjançant el que s'anomena un *transductor* [A3] [A6] [A8], que en aquest cas s'encarrega de transformar el que s'envia pel canal `chunk-chan`. D'aquesta forma, tot el que s'envia per aquest canal serà transformat per tal que si el *tweet* està inacabat s'esperi a tenir-lo complet abans de passar-lo a l'altre extrem del canal. L'elaboració del *transductor* que fa aquesta transformació és una mica complexa i és per això que l'hem reaprofitat del projecte "birdwatch" [R3], desenvolupat per Matthias Nehlsen, el qual també està escrivint un llibre explicant aquest projecte [4]. En el tros de codi següent podem observar la forma com ha estat definit el canal `chunk-chan` per tal d'aplicar el *transductor*.

```
(chan 1 (processing/process-chunk last-received) processing/ex-handler)
```

Codi 4.1-2 Iteració 1: canal amb transductor

La funció `process-chunk` retorna una funció (*transductor*) que serà aplicada a cadascun dels fragments de *tweet* que passin pel canal. El codi d'aquesta funció és el següent:

```
(defn process-chunk
 "Creates composite transducer for processing tweet chunks.
  Last-received atom passed in for updates."
  [last-received]
  (comp
    (streaming-buffer)
    (map json/read-json)
    (filter tweet?)
    (log-count last-received)))
```

Codi 4.1-3 Iteració 1: funció process-chunk

Podem observar que aquesta funció retorna una funció composta per varies funcions. Les veurem en detall.

```
(defn- streaming-buffer []
  (fn [step]
    (let [buff (atom "")]
      (fn
        ([r] (step r))
        ([r x]
          (let [json-lines (-> (str @buff x) (insert-newline)
                               (str/split-lines))
                to-process (butlast json-lines)]
            (reset! buff (last json-lines))
            (if to-process (reduce step r to-process) r)))))))
```

Codi 4.1-4 Iteració 1: funció streaming-buffer

Aquesta funció retorna una primera funció que rep un paràmetre `step`. Aquest paràmetre és la funció que es cridarà amb el tros de *tweet* una vegada processat en aquest pas, és a dir, el següent pas en la cadena de funcions que hem vist que es definien en la funció `process-chunk`. Aquesta primera funció només s'encarrega de declarar un àtom, que s'utilitzarà per emmagatzemar els trossos de *tweet* incomplerts, i retornar una altra funció que pot rebre un o dos paràmetres anomenats `r` i `x`. En aquest cas, el paràmetre `r` serà el tros de *tweet* incomplert rebut en la volta anterior i emmagatzemat dins l'àtom, com veurem posteriorment. D'altra banda, el paràmetre `x` contindrà el tros de *tweet* que estem processant.

En el cas que es cridi la funció amb dos paràmetres, el comportament d'aquesta serà; en primer lloc concatenar ambdues cadenes. Posteriorment, es busca el final del *tweet* i s'aplica un salt de línia en aquest punt en cas que no hi sigui. Finalment, es separen els diferents *tweets* en un vector, en el qual cada posició conté un *tweet* complert. En aquest punt, l'última posició d'aquest vector, la qual podria contenir un *tweet* incomplert, s'emmagatzema en l'àtom mencionat anteriorment. A la resta de *tweets* se'ls aplica la funció `step`, és a dir, el següent pas en la cadena.

En aquest punt, podem estar segurs de que els *tweets* estan sencers. Per tant, el que cal fer és transformar el *tweet*, el qual actualment té un format JSON correcte però és una cadena de caràcters. Mitjançant la funció `read-json`, de la llibreria Clojure.data.json, podem transformar aquesta cadena de caràcters en una taula associativa.

El següent pas s'encarrega de comprovar que el que hem rebut realment és un *tweet* i no un missatge d'estat que envia l'API de Twitter o alguna cosa semblant. Finalment, l'últim pas és un altre *transductor* que en aquest cas s'encarrega d'imprimir missatges de `log` cada 1000 *tweets* processats. A més a més, aquest també actualitza l'àtom que conté el moment en què s'ha processat l'últim *tweet* rebut.

Finalment, ja deixant de banda el procés per obtenir els *tweets*, l'últim que cal fer en el servidor per resoldre la iteració és enviar els *tweets* cap al client mitjançant un *websocket*. La connexió entre servidor i client mitjançant un *websocket* ha estat explicada anteriorment, així que una vegada tenim aquesta connexió feta només cal enviar els *tweets* que rebem pel canal de *tweets* complets, pel *websocket*. Per la qual cosa hem creat la següent funció que envia el *tweet* que se li passa per paràmetre a tots els clients connectats.

```
(defn refresh-all-clients [tweet]
  (doseq [uid (:any @connected-uids)]
    (chsk-send! uid [:tweets/text (:text tweet)])))
```

Codi 4.1-5 Iteració 1: funció refresh-all-clients

Podem veure que utilitzem la funció `doseq`, la qual rep una seqüència i executa el codi que conté per a cadascun dels elements de la seqüència. En aquest punt ens convé fer això per tal d'enviar els *tweets* a cadascun dels clients connectats a l'aplicació.

D'altra banda, aquesta funció serà cridada des de la funció `tweets-loop` la qual podem veure a continuació i que crida la funció anterior passant-li per paràmetre cadascun dels *tweets* que rep pel canal `tweets-chan`.

```
(defn tweets-loop []
  (go-loop [ ]
    (let [tweet (<! tweets-chan)]
      (refresh-all-clients tweet) (recur))))
```

Codi 4.1-6 Iteració 1: funció tweets-loop

Podem observar que utilitzem la funció `go-loop`, la qual és una macro que ho transforma en un bucle dins d'un bloc `go`. El bloc `go` és necessari degut a que dins d'aquest bucle llegim un canal i podria ser que no rebéssim res per ell durant una estona. En cas que no utilitzéssim un bloc `go` el programa es quedaria bloquejat en aquest punt esperant rebre alguna cosa pel canal. En l'apartat **Core.async** hem explicat en més detall els blocs `go`.

4.1.4 Disseny i implementació del client

En primer lloc, en el client ha sigut necessari definir un àtom per a guardar l'estat de l'aplicació i el qual contindrà les dades sobre les quals es construirà la pàgina web.

```
(defonce app-state (atom {:tweets [ ]}))
```

Codi 4.1-7 Iteració 1: estat de l'aplicació

Podem veure que el valor associat a la clau *`tweets`* és un vector buit. No obstant això, en aquest vector s'aniran posant els *tweets* que arribin a l'aplicació fins a un màxim de 10. En aquest moment aquest vector actuarà com una cua FIFO, quedant-nos només amb els 10 últims *tweets* que han arribat a l'aplicació. D'aquest vector s'obtindran els *tweets* que es mostraran en la pàgina web.

Quan a la construcció de la pàgina web, utilitzem Om per mostrar els *tweets* que van arribant a l'aplicació. Com ja hem explicat anteriorment, aquesta llibreria utilitza diferents components per construir la pàgina web. En el nostre cas tenim dos components; el component arrel, el qual crea cadascun dels components de l'aplicació,

i el component de la llista de *tweets*. D'aquesta forma quan vulguem introduir un nou component a la nostra aplicació el crearem des del component general. A més a més, per tal de representar els *tweets* tal com van arribant a l'aplicació, implementem el protocol `IWillMount` d'Om, el qual indica que canviarem alguna cosa de l'estat de l'aplicació. En la implementació d'aquest protocol cridem a la funció següent, la qual s'encarrega de rebre els *tweets* i afegir-los a l'estat de l'aplicació, quedant-se només amb els deu últims.

```
(defn event-loop [cursor owner]
  (go-loop []
    (let [{:keys [event]} (<! ch-chsk)
          [ev-id ev-value] event]
      (if (= :chsk/recv ev-id)
        (let [[_ val] ev-value]
          (when (or (not (string? val)) (not (Clojure.string/blank? val)))
            (Om/transact! cursor [:tweets] #(take 10 (into [val] %)))))))
    (recur)))
```

Codi 4.1-8 Iteració 1: funció event-loop

El primer que podem observar és la crida a la funció `transact!` d'Om. Aquesta funció permet canviar l'estat de l'aplicació, la qual cosa provocarà que Om reconstrueix-hi la pàgina en funció d'aquest nou estat. Utilitzem aquesta funció i no la que utilitzaríem normalment amb un àtom degut a que així Om s'adona que hi han hagut canvis en l'estat de l'aplicació i comprovarà si cal fer canvis en la pàgina. En aquest cas el que estem fent és afegir el nou *tweet* a l'estat de l'aplicació.

Finalment, podem observar que és necessari comprovar que el que rebem pel *websocket* té l'identificador `:chsk/recv` abans de tractar-ho. Això és necessari degut a que el servidor envia periòdicament missatges d'estat pel *websocket* per tal d'informar al client de que el *websocket* segueix obert. A més a més, també hem de comprovar que el *tweet* no és buit i que es tracta d'una cadena, ja que havíem experimentat problemes degut a l'incompliment d'alguna d'aquestes premisses.

La funció encarregada de mostrar els *tweets* és força senzilla i aplica una funció per convertir en un vector amb el format requerit per la llibreria sablono per al contingut html.

```
(defn tweets-view [{:keys [tweets]} owner]
  (reify
    Om/IRender
    (render [_]
      (html
        [:div [:h3 "Recived Tweets:"]
         (map #(vector :p %) tweets)]))))
```

Codi 4.1-9 Iteració 1: funció tweets-view

També podem veure que desestructurem el cursor que es passa a la funció per quedar-nos només amb el contingut de la clau "*tweets*".

4.1.5 Aparença de la pàgina

Figura 10 Aparença de la pàgina després de la Iteració 1

Podem observar que es mostren 10 *tweets*. En arribar un nou *tweet* es mostraria en la part superior i l'últim *tweet* de la part baixa desapareixeria.

4.1.6 Conclusions

En aquesta primera iteració hem pogut experimentar, a banda de les característiques bàsiques d'una pàgina web en Clojure, la connexió entre client i servidor per mitjà d'un *websocket* i la forma de construir una pàgina web que te la llibreria Om.

Cal destacar que no hem tingut gaires problemes per assolir aquesta primera iteració. L'únic que ens ha posat una mica de complicacions han estat els missatges que el servidor envia periòdicament pel *websocket* com a prova de que segueix oberta la connexió, ja que inicialment no sabíem de l'existència d'aquests missatges i teníem un comportament inesperat de l'aplicació en rebre aquests missatges.

Cal tenir en compte que hem necessitat un període de temps considerable per a aquesta iteració degut a que hem començat a desenvolupar-la sense coneixements de la llibreria Om i els hem anat adquirint durant el transcurs d'aquesta a mesura que han sigut necessaris. A més a més, per tal d'adquirir aquests coneixements hem consultat diversos recursos i hem realitzat el tutorial bàsic i de nivell intermig que proporciona Om en la seva pàgina de GitHub.

4.2 Iteració 2

4.2.1 Objectius

En aquesta segona iteració guardarem dades estadístiques de la longitud dels *tweets* i les mostrarem en la pàgina web. Cal destacar que la longitud dels *tweets* és una dada insignificant i molt fàcil de calcular. No obstant això, el vertader objectiu no és conèixer quina és la longitud mitja dels *tweets* que rebem sinó que ens serveix d'excusa per a practicar amb el llenguatge i amb la forma de treballar d'Om.

4.2.2 Refinament dels requeriments

En l'expressió "longitud dels *tweets*" no queda clar si mesurarem per paraules o per caràcters. Per tant, en aquest punt és necessari prendre una decisió al respecte. Així doncs, hem decidit mesurar la longitud en caràcters, ja que ens serà més fàcil calcular-ho mitjançant la funció `count` de Clojure que compta el número de caràcters que té una cadena.

D'altra banda, per tal de poder classificar els *tweets* que ens arriben en funció de la seva longitud, ha sigut necessari definir uns intervals de longitud. Donat que el màxim de caràcters que permet Twitter per a un *tweet* és de 140 caràcters, hem decidit que els intervals siguin els següents:

- Menys de 40 caràcters.
- Entre 41 i 80 caràcters.
- Entre 81 i 120 caràcters.
- Més de 120 caràcters.

4.2.3 Disseny i implementació del servidor

En el servidor no ha sigut necessari realitzar cap canvi respecte al que teníem en acabar la primera iteració, ja que el tractament dels *tweets* per obtenir les dades estadístiques el farem en el client.

4.2.4 Disseny i implementació del client

El primer canvi que hem realitzat en el codi del client ha estat la modificació de l'estat de l'aplicació per tal d'incloure en aquest les dades estadístiques de la longitud dels *tweets*. A més a més, la dada estadística `length` l'hem inclòs dins d'una taula associativa amb clau `statistics`, ja pensant que en un futur podrem afegir altres dades en aquesta taula associativa.

```
(defonce app-state
  (atom {:tweets [ ] :statistics {:length {:-40 0 :40-80 0
                                           :80-120 0 :120+ 0}}}))
```

Codi 4.2-1 Iteració 2: estat de l'aplicació

Podem veure que inicialitzem una parella clau-valor amb el valor de zero per a cada interval. D'aquesta forma, posteriorment incrementarem el valor de la parella corresponent en funció de la longitud del *tweet* que ens ha arribat. Aquest procés el fem en la funció que hem mostrat en l'anterior iteració que s'encarrega de tractar i afegir a l'estat de l'aplicació els *tweets* que rebem.

```
(defn event-loop [cursor owner]
  (go-loop []
    (let [{:keys [event]} (<! ch-chsk)
          [ev-id ev-value] event]
      (if (= :chsk/recv ev-id)
        (let [[_ val] ev-value]
          (if (or (not (string? val)) (not (Clojure.string/blank? val)))
            (let [length (count val)]
              (Om/transact! cursor [:tweets] #(take 10 (into [val] %)))
              (Om/transact! cursor [:statistics :length]
                            #(update-in % [(get-bucket length)] inc)))))))
  (recur)))
```

Codi 4.2-2 Iteració 2: funció event-loop

Podem veure que on hauríem d'indicar la clau de la taula associativa que volem actualitzar cridem la funció `get-bucket`. Aquesta funció ens retornarà la clau de la parella a actualitzar en funció de la longitud del *tweet*, la qual passem per paràmetre a la funció.

En el següent fragment de codi podem observar l'estructura de la funció `get-bucket`. Podem observar que utilitzem la funció `condp` per decidir quina de les claus retornar. Aquesta funció compara el valor del símbol `length` amb les diferents condicions i amb

el comparador que s'indica, en aquest cas ">". La funció retornarà el valor associat amb la primera condició que es compleix-hi. D'aquesta forma, ens retornarà la clau adequada per a cada cas en funció del valor de `length`. Això seria equivalent a fer-ho amb una sèrie de sentències `if-elseif`.

```
(defn get-bucket [length]
  (condp > length
    40 :-40
    80 :40-80
    120 :80-120
    :120+))
```

Codi 4.2-3 Iteració 2: funció get-bucket

A banda d'això, l'únic que ha estat necessari per assolir els objectius de la iteració ha estat implementar una funció que s'encarregui de mostrar les dades estadístiques que es van guardant en l'estat de l'aplicació, la qual té una aparença molt semblant a la que hem vist anteriorment per mostrar els *tweets* que conté l'estat.

```
(defn length-view [cursor owner]
  (reify
    Om/IRender
    (render [_]
      (html
        [:div [:h4 "Number of letters of the tweets:"]
          [:div (str "Less than 40: " (:-40 cursor))]
          [:div (str "From 40 to 80: " (:40-80 cursor))]
          [:div (str "From 80 to 120: " (:80-120 cursor))]
          [:div (str "More than 50: " (:120+ cursor))]])))))
```

Codi 4.2-4 Iteració 2: funció length-view

Finalment, pel mateix motiu que hem explicat pel qual hem afegit un mapa intermig anomenat "estadístiques" en l'estat de l'aplicació, també hem implementat una funció intermitja encarregada de cridar a la mostrada anteriorment. D'aquesta forma, quan

haguem de mostrar una altra dada estadística implementarem una funció com l'anterior amb la nova dada i la cridarem en la funció intermitja.

```
(defn statistics-view [cursor owner]
  (reify
    Om/IRender
    (render [_]
      (html
        [:div [:h3 "Tweets Statistics:"]
          (Om/build length-view (:length cursor))])))))
```

Codi 4.2-5 Iteració 2: funció statistics-view

La funció `build` de la llibreria Om munta un component d'Om en el lloc on es crida.

4.2.5 Aparença de la pàgina

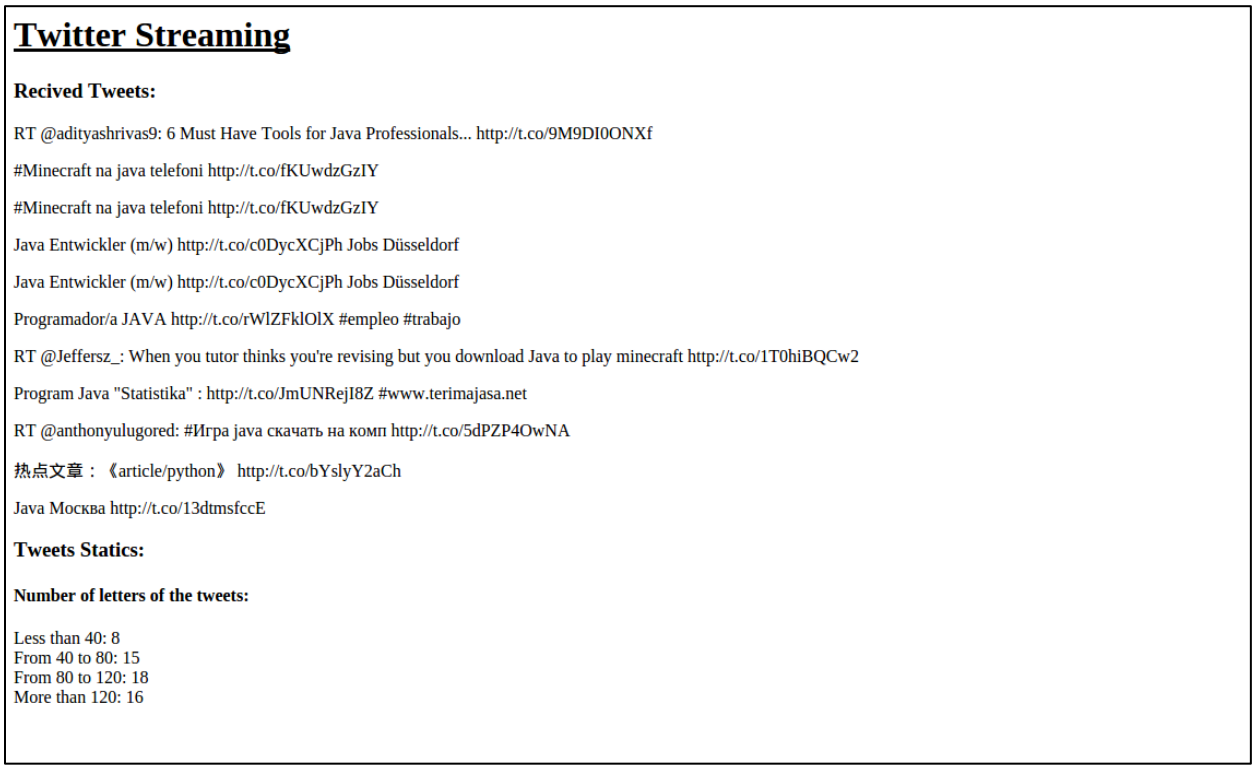


Figura 11 Aparença de la pàgina després de la Iteració 2

A part de la llista de *tweets* vista en la primera iteració, podem observar una segona part de la pàgina en la que es mostra el nombre de *tweets* que hem rebut de cadascun dels intervals de longitud. Aquestes dades s'actualitzen cada vegada que arriba un nou *tweet*.

4.2.6 Conclusions

Amb els coneixements adquirits en la iteració anterior, aquesta segona iteració ha sigut força fàcil i tot ens ha funcionat a la primera, la qual cosa ens permet recuperar temps perdut en la iteració anterior i continuar amb la següent iteració per tal de complir amb els terminis del treball.

4.3 Iteració 3

4.3.1 Objectius

En primer lloc, cal destacar que inicialment en aquesta iteració havíem de satisfer el quart requeriment, el qual indica que hem de mostrar les dades estadístiques calculades sobre la longitud dels *tweets* en forma gràfica. No obstant això, satisfer aquest requeriment implica començar a utilitzar alguna llibreria per a la visualització gràfica de dades, la qual encara no coneixem. És per això que hem decidit intercanviar l'ordre del quart i cinquè requeriments, ja que el cinquè requeriment va en la mateixa línia que la part implementada en la iteració anterior i ja tenim els coneixements per fer la implementació.

D'altra banda, el cinquè requeriment diu que hem d'obtenir dades estadístiques del lloc de procedència dels *tweets* que rebem a l'aplicació. No obstant això, després d'observar la informació que rebem dels diferents *tweets* ens hem trobat amb el problema de que de les dades que obtenim de la API de Twitter no podem obtenir dades sobre la procedència del *tweet*. La única forma d'obtenir aquesta dada seria mitjançant les coordenades i geolocalització, però això seria força complicat i no és

l'objectiu principal del nostre treball. Així doncs, hem optat per canviar la dada sobre la qual obtenir les estadístiques i utilitzarem l'idioma en el qual estan escrits els *tweets*, del qual si que ens arriba informació en cada *tweet*.

L'objectiu principal d'aquesta iteració és guardar dades estadístiques sobre l'idioma dels *tweets* i mostrar aquesta estadística en la pàgina web. No obstant això, com hem pogut observar en les iteracions anteriors, el que enviem al client només és el text del *tweet* i no tota la resta de dades que ens arriben de l'API de Twitter, entre les quals es troba l'idioma. Per tant, aquesta estadística la calcularem en el servidor i enviarem les dades estadístiques periòdicament al client.

4.3.2 Refinament dels requeriments

Tal i com hem esmentat anteriorment, hem hagut de canviar el paràmetre sobre el qual calcular la dada estadística. Per tant, el nou requeriment que substituirà el cinquè dels especificats en la secció "Anàlisi de requeriments" serà el següent:

- "El sistema ha de calcular i guardar estadístiques del llenguatge dels *tweets* per tal de mostrar-les estadísticament."

Cal destacar que en la pàgina web no es mostraran estadístiques de tots els llenguatges amb que ens arribin *tweets*, sinó només dels "n" llenguatges amb major nombre de *tweets*. Per a la resta de *tweets* rebuts en altres idiomes en calcularem el total i mostrarem aquesta dada com a "Altres idiomes". D'altra banda, aquestes dades estadístiques només s'actualitzaran en la pàgina web cada vegada que es rebin un cert nombre de *tweets*, que serà quan el servidor enviarà les dades actualitzades als clients.

4.3.3 Disseny i implementació del servidor

Per realitzar aquesta iteració, el primer que ha sigut necessari ha estat definir un símbol on emmagatzemar les estadístiques de l'idioma dels *tweets* que anem rebent. Aquestes estadístiques les guardarem en una taula associativa i seran parelles "idioma - núm

tweets". D'altra banda, degut a que només ens interessarà quedar-nos amb els "n" idiomes dels que hem rebut més *tweets*, serà convenient que les dades es guardin de forma ordenada directament. Per aconseguir aquest fet, utilitzem la funció `priority-map-by`, la qual retorna una taula associativa en la que automàticament en afegir-hi valors s'ordenaran les parelles d'associacions en funció del valor d'aquestes i comparant-les amb una funció que passarem per paràmetre. Com que en aquest cas els valors d'aquesta estructura només s'utilitzaran en la funció `tweets-loop`, no ha sigut necessari emmagatzemar-ho en un àtom. A canvi, inicialitzarem l'estructura de dades en la definició del bucle i a cada volta d'aquest passarem les dades actualitzades. Així doncs, a partir d'ara, la funció `tweets-loop` també cridarà una altra funció encarregada d'actualitzar les estadístiques dels idiomes en què rebem els *tweets*. A continuació podem veure la nova forma d'aquesta funció.

```
(defn tweets-loop []
  (go-loop [[tick & ticks] (cycle (range (:freq-lang-statistics params)))
            lang-statistics (priority-map-by >)]
    (let [tweet (<! tweets-chan)
          updated-lang-statistics
            (update-language-statistics (:lang tweet) lang-statistics)]
      (refresh-all-clients tweet tick updated-lang-statistics)
      (recur ticks updated-lang-statistics))))
```

Codi 4.3-1 Iteració 3: funció tweets-loop

El primer que cal destacar és que vinculem a un símbol el resultat de la funció `update-language-statistics`, la qual s'encarrega d'actualitzar la taula associativa amb l'idioma de l'últim *tweet* rebut. És per això que a aquesta funció passem com paràmetres l'idioma de l'últim *tweet* i la taula associativa amb les dades que tenim fins al moment. Posteriorment passem aquest símbol com a paràmetre de la següent volta del bucle, ja que conté les dades estadístiques actualitzades.

```
(defn update-language-statistics [lang lang-statistics]
  (update-in lang-statistics [lang] (fnil inc 0)))
```

Codi 4.3-2 Iteració 3: funció update-language-statistics

Donat que, el que hem de fer per actualitzar les dades és incrementar el comptador associat a la clau corresponent a l'idioma que ens arriba per paràmetres, podem fer-ho mitjançant la funció `update-in`. No obstant això, la funció que apliquem sobre el valor associat a la clau no és simplement `inc`, sinó que utilitzem `fnil`, a la qual passem dos paràmetres. Amb la crida d'aquesta funció el que aconseguim és que, en cas que el que s'extragui de la taula associativa en obtenir el valor associat a l'idioma sigui `nil`, s'associï amb aquesta clau el resultat d'aplicar la funció `inc` al segon paràmetre que passem a la funció `fnil`, en aquest cas 0. En cas contrari, aquesta funció no tindrà cap efecte i s'aplicarà la funció `inc` al valor obtingut de la taula associativa. D'aquesta forma aconseguim solucionar el problema en què ens trobaríem en el cas de que fos el primer *tweet* que ens arriba en un determinat idioma, la qual cosa hauríem de tractar diferent degut a que encara no hi hauria una clau per a aquest idioma. Si implementéssim aquesta funció de forma imperativa, tindria el següent aspecte:

```
(defn update-language-statistics [lang lang-statistics]
  (if (contains? lang-statistics lang)
    (update-in lang-statistics [lang] inc)
    (assoc lang-statistics lang 1)))
```

Codi 4.3-3 Iteració 3: funció update-language-statistics (versió imperativa)

D'altra banda, cal destacar l'aparició dels símbol `tick` i `ticks`, els quals ens serveixen per determinar en quin moment enviar les dades estadístiques dels idiomes dels *tweets* als diferents clients. Podem veure que el valor associat a aquests símbols és una llista que creem de forma cíclica en funció de la freqüència en que s'han d'enviar les estadístiques dels idiomes als clients. Per exemple, en cas que haguem d'enviar aquestes estadístiques cada 3 *tweets* rebuts, es crearia la llista infinita següent:

```
(0 1 2 0 1 2 0 1 2 0 1 2 .....)
```

Codi 4.3-4 Iteració 3: Seqüència cíclica infinita de freqüència enviament estadístiques idiomes

En els símbols `tick` i `ticks` desestructurem aquesta llista de manera que en cada volta s'associa a `tick` el primer element de la llista, el qual representa el valor actual, i a `ticks` la llista amb la resta d'elements. D'aquesta forma podem utilitzar el símbol `tick` per determinar quan hem d'enviar les estadístiques als clients, com veurem en la següent funció, i passem el símbol `ticks` a la següent volta del bucle.

n la següent funció podem veure com utilitzem el valor de `tick` per determinar si enviar o no les estadístiques als clients:

```
(defn refresh-all-clients [tweet tick lang-statistics]
  (doseq [uid (:any @connected-uids)]
    (chsk-send! uid [:tweets/text (:text tweet)])
    (if (zero? tick)
      (chsk-send! uid
        [:tweets/lang (get-lang-statistics lang-statistics)]))))
```

Codi 4.3-5 Iteració 3: funció refresh-all-clients

També podem veure que no enviem directament el valor del símbol `lang-statistics` als clients sinó que enviem el que retorna la funció `get-lang-statistics`. Ho fem així perquè no volem enviar totes les estadístiques sinó les dels "n" idiomes dels quals hem rebut més *tweets* i la suma del nombre de *tweets* rebuts en altres idiomes. Això és el que s'encarrega de construir la funció `get-lang-statistics` i ho fa de la següent forma:

```
(defn get-lang-statistics [lang-statistics]
  (let [[lang-statistics other] (split-at (:num-lang-statistics params)
                                          lang-statistics)]
    (conj lang-statistics (apply + (map second other)))))
```

Codi 4.3-6 Iteració 3: funció get-lang-statistics

En aquesta funció utilitzem la funció `split-at` per dividir en dos seqüències diferents les estadístiques dels "n" idiomes amb més *tweets* i les de la resta. Després concatenem les parelles de la primera seqüència obtinguda amb la suma dels valors de la segona, la qual cosa calculem aplicant la suma a la seqüència dels segons elements de cada parella “clau-valor” de la seqüència que conté les estadístiques dels idiomes descartats per ser enviats al client.

Finalment, en les funcions `tweets-loop` i `get-lang-statistics` hem pogut observar que alguns valors els obtenim del símbol `params`, la qual és una taula associativa. Aquests paràmetres són la freqüència en què enviem les dades estadístiques dels idiomes als clients i el número d'idiomes dels que enviem estadístiques als clients. Hem creat un arxiu extern per indicar aquests paràmetres, ja que d'aquesta forma no serà necessari modificar el codi font per canviar-los. Després, obtenim aquests paràmetres en el codi que es mostra a continuació i podem utilitzar el símbol `params` com una taula associativa normal.

```
(def params (edn/read-string (slurp "application-params.edn")))
```

Codi 4.3-7 Iteració 3: símbol params

4.3.4 Disseny i implementació del client

Com hem pogut observar amb els canvis realitzats en el servidor, a partir d'aquest moment enviarem dos tipus de missatges pel *websocket*, un pels *tweets* i un altre per les estadístiques dels idiomes. Per tant, en el client haurem de tractar de forma diferent

el que obtinguem pel *websocket* en funció del tipus de missatge. La forma en què hem solucionat aquest fet resulta força elegant degut a que compleix el principi obert-tancat, el qual diu que el nostre codi ha d'estar obert a possibles extensions però tancat a modificacions.

Per tal d'executar el codi corresponent en funció del que rebem pel *websocket* ho hem fet mitjançant multimètodes. Els quals ens permeten definir diferents funcions que es cridaran en funció d'un valor contingut en els paràmetres de la funció. En el nostre cas, el que rebrem pel *websocket* tindrà la forma següent:

```
{:chsk/recv {:id-tipus-missatge { contingut missatge }}}
```

Codi 4.3-8 Iteració 3: format missatge websocket

Per tant, haurem de decidir cridar una funció o una altra en funció del valor de `:id-tipus-missatge`. Per fer-ho definim la següent funció, la qual s'encarrega de triar el valor que ens servirà per classificar el missatge en la funció corresponent. En aquesta funció desestructurem el format anterior per quedar-nos amb el valor de `:id-tipus-missatge`. Cal tenir en compte que a aquesta funció només passem la part associada a la clau `:chsk/recv`, ja que en la funció `event-loop` ja comprovem que el missatge sigui d'aquest tipus i, per tant, no ens interessa aquesta part.

```
(defmulti handle-event (fn [[type _] _] type))
```

Codi 4.3-9 Iteració 3: definició multi-mètode handle-event

Una vegada definida aquesta funció, el següent que cal fer és definir les diferents funcions que tractaran els missatges. La primera d'elles és l'encarregada de tractar els *tweets* que van arribant pel *websocket* de la mateixa forma que es feia anteriorment.

```
(defmethod handle-event :tweets/text [[_ tweet] cursor]
  (if (or (not (string? tweet)) (not (Clojure.string/blank? tweet)))
    (let [length (count tweet)]
      (Om/transact! cursor [:tweets] #(take 10 (into [tweet] %)))
      (Om/transact! cursor [:statistics :length :data]
                    #(update-in % [(get-bucket length)] inc)))))
```

Codi 4.3-10 Iteració 3: multi-mètode handle-event per clau :tweets/text

La segona funció és l'encarregada de tractar les estadístiques que ens arribaran sobre l'idioma dels *tweets*. En aquesta funció l'únic que cal fer és canviar el valor associat a l'estadística dels idiomes en l'estat de l'aplicació per la nova estadística que acabem de rebre, ja que el càlcul d'aquesta es fa en el servidor. Cal destacar, que en aquest cas no hem modificat l'estructura de l'estat de l'aplicació creant una nova entrada a la taula associativa per a l'estadística dels idiomes, ja que aquesta es crearà automàticament en rebre l'estadística per primera vegada.

```
(defmethod handle-event :tweets/lang [[_ langs] cursor]
 (Om/transact! cursor [:statistics] #(assoc-in % [:langs] langs)))
```

Codi 4.3-11 Iteració 3: multi-mètode handle-event per clau :tweets/lang

D'aquesta forma, també hem aconseguit refactoritzar la funció `event-loop` obtenint un disseny més ordenat.

```
(defn event-loop [cursor owner]
 (go-loop []
   (let [{:keys [event]} (<! ch-chsk)
         [ev-id ev-value] event]
     (if (= :chsk/recv ev-id)
       (handle-event ev-value cursor)))
   (recur)))
```

Codi 4.3-12 Iteració 3: funció event-loop

Finalment, cal crear la funció encarregada de mostrar en la pàgina web l'estadística del idiomes dels *tweets*, que anomenarem `langs-view`. Aquesta funció serà cridada des de la funció `statistics-view`, la qual ja havíem creat en la iteració anterior pensant en aquest moment. Cal destacar que en la funció que mostra les estadístiques dels idiomes comprovarem abans de mostrar cadascuna de les estadístiques si aquesta ens arriba en forma de vector o no. Ja que la suma del número de *tweets* rebuts amb idiomes diferents dels mostrats en les estadístiques ens arriba en un valor individual.

```
(defn langs-view [cursor owner]
  (reify
    Om/IRender
    (render [_]
      (html
        [:div [:h4 "Most used languages of the tweets:"]
          (map (fn [lang-statistic]
                 (if (vector? lang-statistic)
                   [:div (str "Language " (first lang-statistic)
                              ": " (second lang-statistic))]
                   [:div (str "Other languages: " lang-statistic)]))
            cursor)]))))
```

Codi 4.3-13 Iteració 3: funció langs-view

4.3.5 Aparença de la pàgina

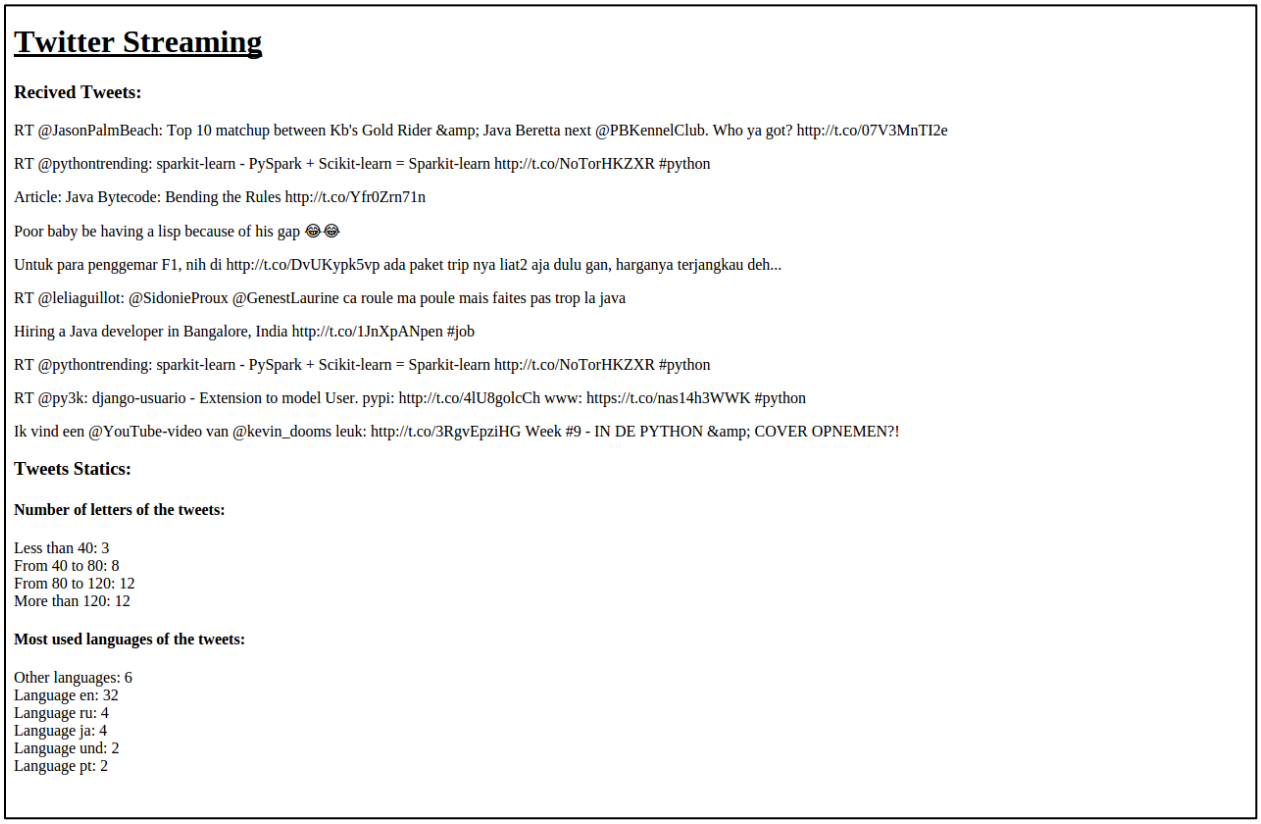


Figura 12 Aparença de la pàgina després de la Iteració 3

Podem observar que a part de les estadístiques calculades en la iteració prèvia, també mostrem estadístiques dels idiomes en els que arriben els *tweets*, la qual cosa era l'objectiu d'aquesta iteració. Aquestes estadístiques s'actualitzen automàticament cada cert nombre de *tweets* rebuts a l'aplicació.

4.3.6 Conclusions

En aquesta iteració ens hem trobat amb el primer problema de caràcter funcional, el qual hem solucionat fàcilment ja que des d'un primer moment hem vist clarament la impossibilitat de complir el requeriment especificat prèviament. Per tant, hem decidit utilitzar una altra característica per calcular l'estadística, ja que el que realment ens

interessava en aquesta iteració era haver de fer el càlcul estadístic en el servidor i passar les dades calculades al client pel *websocket*.

D'altra banda, cal destacar que la part més difícil d'aquesta iteració ha estat el càlcul de la suma de les dades estadístiques dels llenguatges que descartem per ser enviats als clients. Ja que en un primer moment semblava que quedaria un codi una mica robust. No obstant això, al final ho hem implementat d'una forma força elegant. En gran part també gràcies a l'ajuda del tutor del treball, ja que nosaltres encara no estem del tot acostumats a la programació funcional i tendim a implementar de forma imperativa.

Finalment, en aquesta iteració hem introduït els multimètodes. Un tipus de funcions molt útils que ens permeten implementar d'una forma semblant al que seria el polimorfisme en programació orientada a objectes. La qual cosa ens permet, en situacions com la que teníem, fer un codi ordenat i net.

4.4 Iteració 4

4.4.1 Objectius

L'objectiu bàsic d'aquesta iteració és mostrar les estadístiques, calculades a les iteracions 2 i 3, sobre la longitud dels *tweets* i l'idioma en què arriben en forma de gràfica. Substituint així la forma com les mostràvem fins al moment, en forma de text. Cal destacar que inicialment havíem dividit les tasques de representar gràficament cadascuna de les gràfiques en dues iteracions. No obstant això, degut a la semblança d'ambdues tasques i a que, finalment, hem representat totes dues estadístiques en una gràfica de barres. Inicialment no sabíem si utilitzaríem diferent tipus de gràfica, i al fer servir el mateix, hem decidit unir aquestes dues iteracions.

D'altra banda, indirectament, un objectiu molt important d'aquesta iteració també és el d'aprendre a utilitzar una llibreria per a la creació de gràfiques. Serà una llibreria en JavaScript i, per tant, també veurem la forma d'interactuar, des de ClojureScript, amb codi JavaScript.

4.4.2 Refinament dels requeriments

En aquesta iteració, l'únic que cal especificar respecte al requeriment corresponent és el tipus de gràfica que utilitzarem. Aquesta serà una gràfica en forma de diagrama de barres verticals, tant per l'estadística de la longitud dels *tweets* com per la de l'idioma dels *tweets*.

4.4.3 Disseny i implementació del servidor

La part del servidor l'hem deixat tal i com estava en la iteració prèvia. No obstant això, donat que per tal de representar les dades en forma gràfica és necessari canviar una mica el format en què les hem emmagatzemat, en un primer moment vam pensar en fer aquesta transformació en el servidor abans de transmetre les dades sobre l'idioma dels *tweets* cap al client. Finalment però, vam creure que no era el servidor qui s'havia d'adaptar a les necessitats del client sinó a la inversa, ja que habitualment el servidor emmagatzema les dades en un format el més neutre possible i són els clients els que les adapten a les seves necessitats.

4.4.4 Disseny i implementació del client

Donat que el que volem fer en aquesta iteració és mostrar gràficament les dades que ja tenim en el client, només serà necessari modificar aquest per tal d'aconseguir-ho. Per tant, el primer que haurem de fer és una funció que, donades unes dades que li arribin per paràmetre, ens les mostri en un gràfic de barres. Per fer aquesta tasca utilitzarem la llibreria de JavaScript Dimple.js [W5], la qual utilitza D3.js per formar les gràfiques però treballa a més alt nivell, facilitant així la tasca als programadors. És per això que en la pàgina "index.html" indicarem que utilitzarem aquestes llibreries de la següent forma:

```
<script type="text/javascript" src="http://d3js.org/d3.v3.min.js"></script>
<script type="text/javascript"
 src="http://dimplejs.org/dist/Dimple.js.v2.0.0.min.js"> </script>
```

Codi 4.4-1 Iteració 4: referència a llibreries D3.js i Dimple.js en pàgina html

En aquest punt ja podem implementar la funció que crearà la gràfica. Cal destacar que per implementar aquesta funció ens hem basat en el projecte "pumpkin" d'Ana Pawlicka [R9], el qual utilitza per explicar com crear gràfiques en una de les seves conferències [P4], la qual també ens ha resultat molt útil per entendre el seu funcionament.

```
(defn- draw-chart [data div
                   {:keys [id bounds x-axis y-axis plot series color]}]
 (let [width        (or (:width div) (:width (default-size id)))
       height       (or (:height div) (:height (default-size id)))
       data         data
       Chart        (.-chart js/Dimple.js)
       svg          (.newSvg js/Dimple.js (str "#" id) width height)
      Dimple.js-chart (.setBounds (Chart. svg) (:x bounds) (:y bounds)
                       (:width bounds) (:height bounds))
       x            (doto (.addCategoryAxis Dimple.js-chart "x" x-axis)
                      (aset "title" nil))
       y            (doto (.addMeasureAxis Dimple.js-chart "y" y-axis)
                      (aset "title" nil))
       s            (.addSeries Dimple.js-chart series plot (clj->js [x y]))
       color-fn     (-> js/Dimple.js .-color)]
   (aset s "data" (clj->js data))
   (aset Dimple.js-chart "defaultColors" (to-array [(new color-fn color)]))
   (.addOrderRule x (sort-by-field y-axis))
   (.draw Dimple.js-chart)))
```

Codi 4.4-2 Iteració 4: funció draw-chart

En primer lloc, podem veure que la funció rep tres paràmetres, un dels quals és una taula associativa, mitjançant els quals crearem la gràfica:

- **data:** conté les dades que seran representades en la gràfica. El format d'aquestes serà una seqüència de llistes associatives, el format de les quals és {:valor-eix-x "Eix X" :valor-eix-y Y}. Per tant, amb cadascuna d'aquestes llistes associatives es crearà una barra del gràfic.
- **div:** aquesta símbol conté una taula associativa que conté l'amplada i alçada la parcel·la on situarem el gràfic.
- **Taula associativa:**
    - **id:** identificador del `div` HTML del qual penjarem la gràfica.
    - **bounds:** conté quatre parelles clau-valor que indiquen les longituds dels eixos de la gràfica i l'altura i amplada del gràfic.
    - **x-axis:** nom de la clau "valor-eix-x" en les llistes associatives del símbol data.
    - **y-axis:** nom de la clau "valor-eix-y" en les llistes associatives del símbol data.
    - **plot:** indica el tipus de gràfica que es construirà.
    - **series:** arriba amb valor `null`, ja que no és necessari utilitzar-lo.
    - **color:** color de les barres de la gràfica.

D'altra banda, donat que la llibreria Dimple.js està implementada en JavaScript i nosaltres treballem en ClojureScript, hem d'utilitzar una notació una mica diferent. Per exemple, en JavaScript podríem accedir als elements d'un objecte de la forma `obj.element`, amb el qual obtindríem el contingut d'aquest element. En canvi, donat que aquest element podria ser una funció, també podem cridar l'element de l'objecte, de manera que s'executaria la funció. Això es faria en JavaScript de la forma `obj.element(paràmetres)`. En canvi, en ClojureScript no podem fer-ho així i és necessari una forma diferenciadora de la forma en què volem obtenir un element. Per al cas d'obtenir l'element de l'objecte sense executar-lo es fa de la forma `(.-element obj)`. I per al cas de voler executar la funció que conté l'element ho fem mitjançant la forma `(.element obj param1 param2 ...)`. A més a més, també és necessari fer una transformació de les estructures de dades creades en ClojureScript abans de passar-les per paràmetres a funcions en JavaScript, ja que és representen de forma

diferent en memòria. Per fer aquesta transformació disposem de la funció `clj->js`, a la qual passem per paràmetre l'estructura que volem transformar i ens la retorna en format adequat per ser processat en JavaScript.

També és necessari conèixer el que és un SVG [W7], ja que Dimple.js ho utilitza per a la creació de la gràfica. Un SVG (Scalable Vector Graphic) és una especificació per descriure gràfics vectorials bidimensionals en format XML. A més a més, aquesta forma de crear gràfics té la característica de ser redimensionables, de manera que en ampliar o reduir el mida del gràfic no es veurà pixelat, sinó que conservarà la seva forma i nitidesa.

Amb tot això, ja serem capaços de crear una gràfica utilitzant la llibreria Dimple.js. Per fer-ho, és necessari crear un objecte de tipus `Chart` a partir del SVG que ens crearà automàticament Dimple.js i el penjarà de l'objecte html amb l'id que li arribarà com a paràmetre. Després, utilitzarem els mètodes de l'objecte de tipus `Chart` per indicar les diferents característiques que volem que tingui el gràfic, les quals aniran en funció dels paràmetres que rep la funció. En aquest punt hem de destacar la forma que hem utilitzat per eliminar els títols dels eixos de les gràfiques, la qual cosa fem mitjançant la funció `doto` en el mateix moment de crear els eixos. Aquesta funció permet crear un objecte JavaScript i aplicar-li una funció o, com en aquest cas, donar valor a un atribut de l'objecte creat. La qual cosa en JavaScript faríem de la següent forma:

```
var y = Dimple.js-chart.addMeasureAxis("y", y-axis);
y.title = "";
```

Codi 4.4-3 Iteració 4: canvi títol eix gràfica en JavaScript

Finalment, cridarem el mètode `draw` de l'objecte de tipus `Chart`, el qual ens crearà el gràfic en funció de les característiques indicades. D'altra banda, podem observar que en aquesta funció es criden les funcions `default-size` i `sort-by-field`. La primera d'elles retorna una taula associativa indicant un valor per defecte per a l'alçada i amplada de la gràfica. La segona funció retorna una funció encarregada de comparar

dos camps de la gràfica per tal de decidir quin representar abans i quin després, per tal d'ordenar-los de major a menor.

En segon lloc, és necessari implementar un component Om per crear i actualitzar les gràfiques quan sigui convenient. Aquesta funció serà l'anomenada `bar-chart`, la qual mostrem a continuació:

```
(defn bar-chart
 "Simple bar chart done using dimple.js"
 [{:keys [data div]} owner {:keys [id] :as opts}]
 (reify
   Om/IWillMount
   (will-mount [_]
     (.addEventListener js/window "resize"
       (fn []
         (let [e (.getElementById js/document id)
               x (.-clientWidth e)
               y (.-clientHeight e)]
           (Om/update! div {:width x :height y})))))
   Om/IRender
   (render [_]
     (html
      [:div {:id id :style {:height "90%"}}]))
   Om/IDidMount
   (did-mount [_]
     (draw-chart data div opts))
   Om/IDidUpdate
   (did-update [_ _ _]
     (let [n (.getElementById js/document id)]
       (while (.hasChildNodes n)
         (.removeChild n (.-lastChild n))))
     (draw-chart data div opts))))
```

Codi 4.4-4 Iteració 4: funció bar-chart

Podem observar que aquesta funció implementa quatre protocols d'Om. En el primer d'ells, `IWillMount`, el que fem és posar un `ActionListener` que s'encarregarà de redimensionar la gràfica en el moment en què es canviïn les dimensions de la finestra del navegador. El segon protocol és el que implementem habitualment, `IRender`. Sorprèn en aquest cas que en aquest punt l'únic que fem és crear un `div` identificat per l'identificador que rebem per paràmetres. Això és perquè tant Om com Dimple.js estan preparats per prendre el control sobre els elements del `DOM` que construeixen però, en aquest cas, ambdós xocarien si construíssim la gràfica en aquest punt degut a que totes dues llibreries voldrien prendre control sobre elles. Per evitar aquesta disputa el que fem és construir la gràfica en el `div` que hem creat, però en els protocols `IDidMount` i `IDidUpdate`, és a dir, després de muntar o actualitzar les dades del cursor que arriben al component. A més a més, per desactivar totalment el control de Dimple.js sobre les gràfiques el que fem és eliminar-les i tornar-les a crear cada vegada que s'actualitzen les dades. No obstant això, podem observar que la tasca d'eliminar els elements que pengen del div on volem construir la gràfica només ho fem en el protocol `IDidUpdate`, ja que el protocol `IDidMount` només s'executa una vegada immediatament després del protocol `IRender` i, per tant, no pot haver res en el `div` que s'acaba de crear.

D'altra banda, com ja hem esmentat anteriorment, és necessari canviar el format en el que tenim les dades que s'hauran de representar en les gràfiques. El format de les dades ha de ser una seqüència de llistes associatives en el formar `{:valor-eix-x "Eix X" :valor-eix-y Y}`. Per això hem creat les següents funcions que s'encarreguen de transformar les dades abans de ser passades per paràmetres a la funció que crea les gràfiques.

```
(defn- reformat-lang [lang]
 (if (vector? lang)
   (let [[lang-key count] lang]
     {:language (or ((keyword lang-key) langs-traduction)) :count count})
   {:language "Other" :count lang}))

(defn- reformat-length [[length count]]
 {:length length :count count})
```

Codi 4.4-5 Iteració 4: funcions reformat-langs i reformat-length

Cal destacar que en el cas de la longitud dels *tweets* no hem canviat directament el format en què guardem les dades tal com ens van arribant, ni tampoc ho canviem directament sobre l'estat en fer la transformació, ja que per actualitzar les dades quan ens arriben nous *tweets* ens és més útil el format en què tenim les dades en l'estat. En canvi, en el cas de les estadístiques de l'idioma en què arriben els *tweets* la transformació l'apliquem en el moment en què ens arriben les estadístiques calculades del servidor, ja que en el client només les utilitzem per crear les gràfiques. A més a més, en aquesta funció podem veure que fem una transformació de l'idioma que ens arriba des del servidor. Això ho fem perquè el format en què ens arriba l'idioma des del servidor és en forma de dos lletres, per exemple "es". No obstant això, és interessant que en la interfície gràfica es mostri l'idioma complet. És per això que hem creat una taula associativa que conté totes les transformacions d'idiomes i en aquest punt fem la transformació en base a aquesta llista.

Finalment, donat que ja estem en un punt força avançat del projecte i la pàgina web comença a tenir varis components, hem aprofitat aquesta iteració per introduir una mica de CSS per ordenar els components de la pàgina. Per fer-ho hem utilitzat Bootstrap i l'únic que hem hagut de fer ha estat referenciar les llibreries en la pàgina HTML, igual que hem fet amb Dimple.js i D3.js, i afegir la classe corresponent als elements HTML que hem anat creant. En aquest cas, el que volíem fer era dividir els elements de la pàgina en dues columnes i la classe que hem utilitzat ha estat `col-md-6`. A més a més, també

hem utilitzat la classe `text-center` per centrar el títol de la pàgina. Podem veure un exemple de com utilitzem aquests elements de Bootstrap en el següent exemple.

```
(defn application [cursor owner]
 (reify
   Om/IWillMount
   (will-mount [_]
     (event-loop cursor owner))
   Om/IRender
   (render [_]
     (html
       [:div
         [:h1 {:class "text-center"} "Twitter Streaming"]
         [:div {:class "col-md-6"} (Om/build tweets-view cursor)]
         [:div {:class "col-md-6"}
               (Om/build statistics-view (:statistics cursor))]])))))
```

Codi 4.4-6 Iteració 4: funció aplication

4.4.5 Aparença de la pàgina

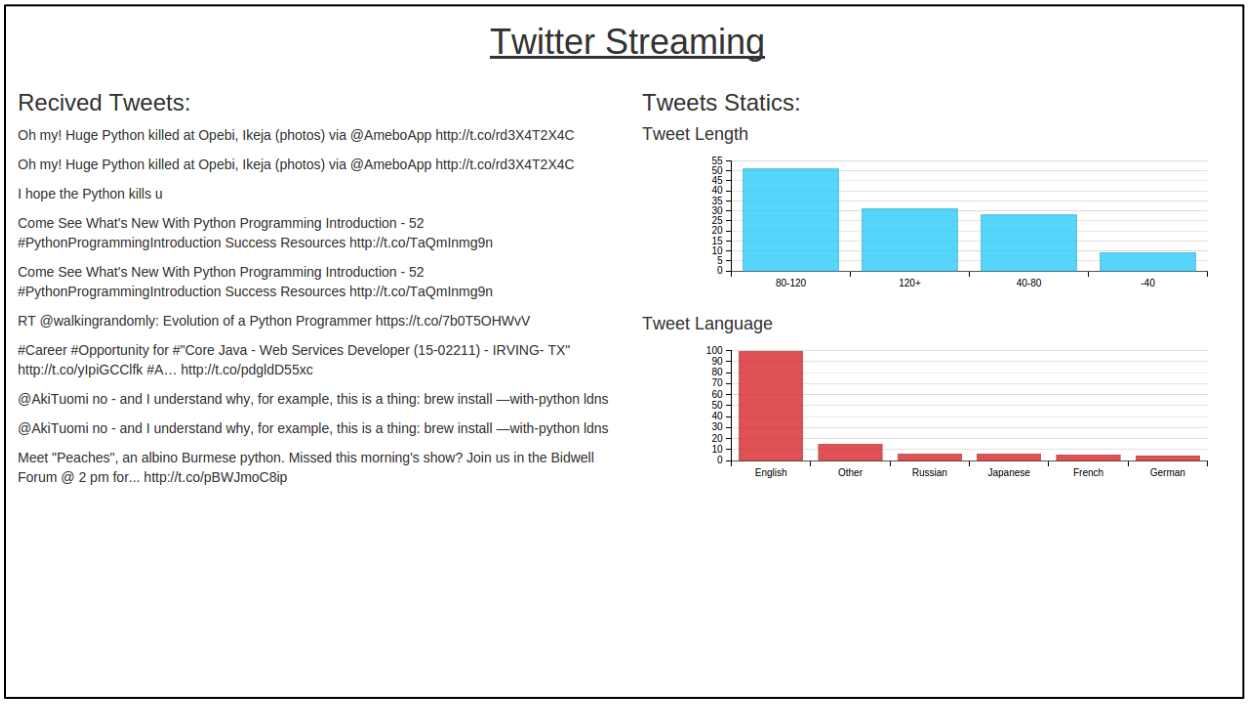


Figura 13 Aparença de la pàgina després de la Iteració 4

En aquesta iteració hi ha hagut un gran canvi en el contingut de la pàgina degut a que hem introduït "Bootstrap" per tal de mostrar el contingut en dues columnes. A més a més, les estadístiques que hem calculat en les iteracions anteriors i que mostràvem en forma de text, les mostrem ara mitjançant dues gràfiques de barres.

4.4.6 Conclusions

En aquesta iteració hem començat a utilitzar la llibreria Dimple.js per a construir gràfiques en representació de les dades estadístiques que havíem calculat en les iteracions prèvies. Per tal d'iniciar-nos en la utilització de la llibreria Dimple.js ho hem fet mitjançant dos projectes d'exemple d'Ana Pawlicka [R9] [R9], els quals explica detalladament en dues conferència [P4] [P5]. En base a aquest projecte hem començat a construir les nostres gràfiques, de manera que l'únic que hem fet ha sigut aplicar la "plantilla" obtinguda del projecte d'exemple i canviar el necessari fins a obtenir el format

que nosaltres volem. A més a més, hem hagut d'esbrinar el format en què hem de passar les dades a la funció que construirà la gràfica per tal de que la pugui construir correctament.

Per tant, tot i que els canvis reflectits en el projecte han sigut força notoris després d'aquesta iteració, ha estat una iteració més d'investigació i aprenentatge, tant de la llibreria Dimple.js com de la notació que s'utilitza per utilitzar codi JavaScript des de ClojureScript.

4.5 Iteració 5

4.5.1 Objectius

En aquesta iteració afegirem un camp d'entrada per tal de permetre a l'usuari introduir un número per indicar el número d'idiomes diferents que vol veure representats en la gràfica.

Aquesta nova funcionalitat de la pàgina ens servirà per experimentar amb la transmissió d'informació de l'aplicació des dels clients cap al servidor, cosa que fins ara no havíem necessitat fer, tot i que ho hem estat fent involuntàriament, ja que automàticament la llibreria sente envia missatges dels clients cap al servidor en obrir el *websocket*. També podrem veure com es crea i es tracta un petit formulari en ClojureScript. A més a més, haurem de resoldre el problema d'haver de tenir controlats cadascun dels clients connectats al servidor, ja que haurem d'enviar-los diferents estadístiques dels *tweets* rebuts de cada idioma en funció del que hagin indicat en el camp d'entrada.

D'altra banda, també volem experimentar una mica més amb la llibreria Dimple.js. És per això que hem cregut oportú modificar la gràfica que representa la longitud dels *tweets* rebuts per tal de que sigui una gràfica de barres horitzontals. També hem pensat que en la gràfica dels idiomes dels *tweets* quedaria millor si la barra que representa l'etiqueta "Altres" es mostrés la última de totes. Així que un altre dels objectius

d'aquesta iteració és implementar aquesta millora, la qual cosa també ens servirà per acabar de veure el funcionament d'un altre dels aspectes de la llibreria.

4.5.2 Refinament dels requeriments

En aquesta iteració hauríem d'implementar els requeriments 7 i 8, definits en l'apartat **Requeriments inicials**. No obstant això, degut a la complexitat que suposaria satisfer aquests requeriments hem decidit canviar-los per una cosa més assequible i adaptats a les nostres restriccions temporals. Així doncs, el requeriment que satisfarem en aquesta iteració és el següent:

> *El sistema permetrà indicar el número d'idiomes dels quals vol, cada client, rebre estadístiques.*

Cal destacar que el servidor enviarà les estadístiques del número d'idiomes indicats al client corresponent immediatament després que aquest premi el botó per seleccionar el número d'idiomes. En cas de no fer-ho així, el client no veuria els canvis reflectits en la pàgina fins que el servidor tornés a enviar les dades estadístiques. Amb la qual cosa podria semblar que no ha tingut cap efecte haver indicat el nombre d'idiomes. Com se'ns va explicar en l'assignatura "Interacció Persona-Ordinador", donar respostes immediates als usuaris després de que aquests realitzin alguna acció és molt important, ja que, en cas contrari, estaríem creant confusió a l'usuari, que es preguntaria si la seva acció ha tingut algun efecte.

D'altra banda, en aquesta iteració també hem de modificar una mica el requeriment 4 de l'anàlisi de requeriments inicial per tal d'indicar el tipus de gràfica que volem per a representar la longitud dels *tweets*. Així doncs, el requeriment serà el següent:

> *El sistema ha de mostrar en la pàgina web una gràfica de barres horitzontals en la que es representi l'estadística de la* longitud *dels tweets.*

4.5.3 Disseny i implementació del servidor

A partir d'ara, el servidor també haurà de tractar el que rebi pel *websocket*. És per això que el primer que ha sigut necessari fer, ha estat crear una funció que, mitjançant un `go-loop`, romandrà tota l'estona esperant rebre alguna cosa pel *websocket*.

```
(defn event-loop []
 (go-loop []
   (let [{:keys [event]} (<! ch-chsk)]
     (thread (handle-event event)))
   (recur)))
```

Codi 4.5-1 Iteració 5: funció event-loop

Podem veure que utilitzem la funció `thread` per tal de fer el tractament del qual hem rebut pel *websocket* en un fil d'execució diferent. D'aquesta forma podríem tractar varis events simultàniament. A diferència del que passa en ClojureScript, aquí sí disposem de fils que poden executar-se concurrentment. No obstant això, la funció `thread` no crea un fil cada vegada que se la crida sinó que agafa els fils d'un `thread pool`. En cas contrari, seria perillós carregar el servidor de massa fils, la qual cosa afectaria el rendiment d'aquest.

D'altra banda, utilitzem multimètodes per tractar els events que ens arribin pel *websocket*. La idea és fer el mateix que hem fet en el client per a tractar els missatges que ens arribaven des del servidor. Cal destacar que actualment només tenim un event possible, ja que només hi ha una funcionalitat que requereix-hi transmissió de missatges pel *websocket* en sentit client-servidor. No obstant això, ho implementem d'aquesta manera per deixar-ho preparat de cara a possibles noves funcionalitats futures. A més a més, fer-ho d'aquesta forma també ens ajuda a evitar possibles problemes en desestructurar un event que no té la forma que esperem, ja que en alguns casos els clients envien missatges per indicar que encara estan connectats al *websocket* i amb la utilització dels multimètodes aquests s'ignorarien automàticament.

Així doncs, la funció encarregada de tractar l'event que arribi pel *websocket* indicant el número d'idiomes dels quals vol rebre informació estadística un determinat client és la següent:

```
(defmethod handle-event :twitter_websockets/langs-count
  [[_ {:keys [nlangs uid]}]]
 (swap! num-langs-clients assoc uid (read-string nlangs))
 (chsk-send! uid [:tweets/lang
                  (get-lang-statistics @lang-statistics
                    (get-num-langs-uid @num-langs-clients uid))]))
```

Codi 4.5-2 Iteració 5: multi-mètode handle-event per clau :twitter_websockets/langs-count

Podem veure que el primer que fem és modificar l'àtom `num-langs-clients`, el qual conté una taula associativa. A aquesta llista guardem (o modifiquem) la parella clau-valor on la clau és l'`uid` del client, el qual hem inclòs en el que transmetem pel *websocket*, i el valor és el número d'idiomes dels quals vol rebre estadístiques aquest client concret. Després, cada vegada que enviem estadístiques a un client, comprovarem en aquest àtom si ha indicat aquest número i, en cas de no haver-ho fet, agafarem el valor per defecte que s'indica en el fitxer de configuracions.

El segon que fem és enviar les dades estadístiques al client per tal que se li mostrin les estadístiques del número d'idiomes seleccionats immediatament després que l'usuari premi el botó per enviar el número d'idiomes que vol veure representats en la gràfica. En cas de no fer-ho d'aquesta forma, no s'actualitzarien les dades fins que el servidor tornés a enviar-les a tots els clients connectats i crearia confusió a l'usuari en no veure cap canvi en l'aplicació fins al cap d'una estona.

Cal destacar que, tal i com podem veure en el fragment de codi anterior, tenim un àtom (`lang-statistics`) on emmagatzemem les estadístiques de l'idioma en què ens arriben els *tweets*. En canvi, en iteracions anteriors aquestes estadístiques les emmagatzemàvem en un símbol local de la funció encarregada de processar els *tweets* i enviar-los als clients. No obstant això, hem decidit canviar-ho per tal de poder enviar les estadístiques al client en el moment en què rebem el número d'idiomes dels que vol

rebre l'estadística, ja que en cas contrari no tindríem accés a les estadístiques dels idiomes des d'aquest punt.

Així doncs, una vegada aplicats aquests canvis, el codi de la funció `tweets-loop`, encarregada de processar els *tweets* que arriben de l'API de Twitter per actualitzar les dades estadístiques i enviar els *tweets* als clients, queda de la següent forma:

```
(defn tweets-loop []
 (go-loop [[tick & ticks] (cycle (range (:freq-lang-statistics params)))]
   (let [tweet (<! tweets-chan)]
     (update-language-statistics (:lang tweet))
     (refresh-all-clients tweet tick)
     (recur ticks))))
```

Codi 4.5-3 Iteració 5: funció tweets-loop

Podem veure que el que ha canviat respecte al codi que havíem implementat en la iteració 3 és que hem suprimit tot el referent al símbol local on emmagatzemàvem les dades estadístiques, ja que ara les tenim en un àtom. Així doncs, les actualitzacions de les dades estadístiques es faran sobre aquest àtom. Ho podem veure en el codi de la funció `update-language-statistics`:

```
(defn update-language-statistics [lang]
  (swap! lang-statistics #(update-in % [lang] (fnil inc 0))))
```

Codi 4.5-4 Iteració 5: funció update-language-statistics

A més a més, la funció `refresh-all-clients` també ha canviat una mica, ja que abans també utilitzava les dades de la variable local i ara emprarà les de l'àtom.

```
(defn refresh-all-clients [tweet tick]
 (doseq [uid (:any @connected-uids)]
   (chsk-send! uid [:tweets/text (:text tweet)])
   (if (zero? tick)
     (chsk-send! uid [:tweets/lang
                      (get-lang-statistics @lang-statistics
                        (get-num-langs-uid @num-langs-clients uid))]))))
```

Codi 4.5-5 Iteració 5: funció refresh-all-clients

Finalment, podem veure que el segon paràmetre que passem a la funció `get-lang-statistics` és el resultat de la funció `get-num-langs-uid`, a la qual passem com a paràmetres el valor de l'àtom `num-langs-clients` i l'`uid` del client al que anem a enviar les dades estadístiques. Aquesta funció s'encarrega de retornar el valor associat a l'`uid` en la taula associativa continguda en l'àtom que també passem com paràmetre o, en el seu defecte, el valor que tenim en el fitxer de configuracions tal i com ho fèiem en les iteracions prèvies. Per fer-ho, la funció `get` permet indicar un valor per defecte que serà retornat en cas que no existeix-hi la clau que indiquem en la taula associativa.

```
(defn get-num-langs-uid [langs-clients uid]
 (get langs-clients uid (:num-lang-statistics params)))
```

Codi 4.5-6 Iteració 5: funció get-num-langs-uid

4.5.4 Disseny i implementació del client

Per satisfer els requeriments d'aquesta iteració hem hagut de fer tres canvis ben diferenciats, un per a cadascun dels objectius de la iteració.

4.5.4.1 Gràfica estadística de la longitud dels *tweets*

Per tal que la gràfica que mostra les estadístiques de la longitud dels *tweets* el primer que hem tingut que fer ha estat duplicar la funció encarregada de crear les gràfiques, que fins al moment era una sola funció que creava les gràfiques en funció dels paràmetres que rebia. Ara no podem fer-ho així degut a que per crear la gràfica de barres horitzontal hem de fer que l'eix de les x sigui "categòric" i el de les y sigui "numèric". En canvi, quan la gràfica és de barres verticals el tipus d'eixos és al contrari. Així doncs, la nova funció per crear aquesta gràfica és la següent:

```
(defn- draw-horizontal-chart [data div {:keys [id bounds x-axis y-axis
                                               plot series color]}]
 (let [width          (or (:width div) (:width (default-size id)))
       height         (or (:height div) (:height (default-size id)))
       data           data
       Chart          (.-chart js/Dimple.js)
       svg            (.newSvg js/Dimple.js (str "#" id) width height)
       Dimple.js-chart (.setBounds (Chart. svg) (:x bounds) (:y bounds)
                                   (:width bounds) (:height bounds))
       x              (doto (.addMeasureAxis Dimple.js-chart "x" x-axis)
                            (aset "title" nil))
       y              (doto (.addCategoryAxis Dimple.js-chart "y" y-axis)
                            (aset "title" nil))
       s              (.addSeries Dimple.js-chart series plot
                                                  (clj->js [x y]))
       color-fn     (-> js/Dimple.js .-color)]
   (aset s "data" (clj->js data))
   (aset Dimple.js-chart "defaultColors" (to-array [(new color-fn color)]))
   (.draw Dimple.js-chart)))
```

Codi 4.5-7 Iteració 5: funció draw-horizontal-chart

Cal destacar que també hem invertit els valors dels eixos, ja que ara l'eix de les x serà la longitud i el de les y contindrà les franges. Aquest canvi l'hem fet en el component Om en el qual indiquem els paràmetres de la gràfica.

```
(defn langs-view [{:keys [title data div] :as cursor} owner]
 (reify
   Om/IRender
   (render [_]
     (html
       [:div [:h4 title]
        (Om/build charts/bar-chart {:data data :div div}
                  {:opts {:id "langs-chart"
                          :bounds {:x "7%" :y "5%"
                                   :width "90%" :height "80%"}
                          :x-axis "language"
                          :y-axis "count"
                          :plot js/Dimple.js.plot.bar
                          :color "#d62728"}})
        (Om/build post-form cursor)]))))
```

Codi 4.5-8 Iteració 5: funció langs-view

4.5.4.2 Etiqueta "Altres" en última posició en la gràfica de l'estadística dels idiomes

En la iteració anterior ordenàvem les diferents etiquetes segons el número de *tweets* rebuts en l'idioma corresponent de major a menor. Això ho fèiem mitjançant una funció que rep les dades de dues de les barres que es crearan i indica quina de les dues s'ha de mostrar primer. Per tant, el que hem fet ha estat modificar aquesta funció per tal que en cas que una de les dades que arriben a la funció sigui la corresponent a l'etiqueta "Altres", la funció sempre indiqui que l'altra barra s'ha de mostrar primer.

```
(defn sort-by-field [field]
 (fn [x y]
   (let [v1 (if (= (str (aget x "language")) "Other")
              0
              (first (aget x field)))
         v2 (if (= (str (aget y "language")) "Other")
              0
              (first (aget y field)))]
     (- v2 v1))))
```

Codi 4.5-9 Iteració 5: funció sort-by-field

Aquesta funció el que fa és comparar els valors de les barres i retorna i retorna -1, 0 o 1 per indicar quin és major i, per tant, s'ha de mostrar primer. Sabent això, hem resolt el nostre objectiu comprovant si alguna de les dades que ens han arribat correspon a l'etiqueta "Altres" i, en aquest cas, forçar que el valor associat a aquesta barra sigui 0. D'aquesta forma, seguint el comportament normal de la funció, sempre indicarà que les altres dades tenen un valor major i, per tant, s'han de mostrar abans.

4.5.4.3 Enviar número d'idiomes dels quals volem veure'n les estadístiques

Per tal de permetre als usuaris introduir un valor per indicar el nombre d'idiomes dels quals volen veure'n les estadístiques el primer que hem hagut de fer ha estat crear un component d'Om on creem l'html per mostrar el text descriptiu del camp d'entrada, el propi camp d'entrada i un botó per fer efectiu el valor introduït en aquest. Cal destacar que la forma com hem estructurat aquesta funcionalitat pot semblar una mica estranya. Ho hem fet d'aquesta manera seguint l'exemple de formulari que consta en el tutorial bàsic d'Om [R4], el qual ja ens havia servit anteriorment per començar a entendre la llibreria.

```
(defn num-langs-form [_ owner]
 (reify
   Om/IInitState
   (init-state [_]
     {:num-langs ""})
   Om/IRenderState
   (render-state [_ state]
     (html
       [:div
        [:label.col-xs-6 "Number of languages to show statistics:"]
        [:div.col-xs-2
         [:input.form-control {:type "text" :ref "num-langs"
                              :value (:num-langs state)
                              :onChange #(handle-change % owner state)}]]
        [:div.col-xs-2
         [:button.btn {:type "button"
                       :on-click #(notify-server owner)} "Select"]]])))))
```

Codi 4.5-10 Iteració 5: funció num-langs-form

El primer que podem observar és que fem ús de l'estat del component d'Om, en ell guardarem en tot moment el que hi hagi en el camp d'entrada. És per això que en la definició del camp d'entrada obtenim que el valor d'aquest camp de l'estat. A més a més, apliquem una funció a l'event onChange per tal que s'executi cada cop que canviï el camp.

```
(defn handle-change [e owner {:keys [num-langs]}]
 (let [value (.. e -target -value)]
   (if-not (re-find #".*\D.*" value)
     (Om/set-state! owner :num-langs value)
     (Om/set-state! owner :num-langs num-langs))))
```

Codi 4.5-11 Iteració 5: funció handle-change

En aquesta funció rebem l'event que ha generat la crida d'aquesta i d'ell podem obtenir el valor del camp d'entrada. Per a obtenir-lo hem d'agafar l'atribut `target` i, dins d'aquest, obtenir el `value`. Per fer-ho utilitzem la macro de Clojure "`..`", que el que fa seria `(. (. e -target) -value)`. És a dir, obtenim el `target` de l'event i d'aquest obtenim el `value`.

Una vegada ja tenim el valor actual del camp d'entrada la cosa és més simple. Utilitzem la sentència `if-not` per comprovar que el valor del camp no conté cap caràcter que no sigui un dígit. Després actualitzem l'estat del component amb el nou valor o el que hi havia anteriorment, en funció de si el nou valor conté algun caràcter no dígit, ja que en aquest camp només té sentit posar-hi un valor numèric. Així doncs, serà després de l'execució d'aquesta funció quan es canviarà el valor que apareix en el camp d'entrada, ja que aquest mostra sempre el valor contingut en l'estat del component.

Finalment, cal veure la funció `notify-server`, la qual es crida en prémer el botó creat en al funció `num-langs-form`. Aquesta funció l'únic que té que fer és obtenir el número indicat en el camp d'entrada i enviar-lo, juntament amb l'`uid` del client, pel *websocket* cap al servidor.

```
(defn notify-server [owner]
 (let [nlangs (-> (om/get-node owner "num-langs") .-value)]
   (when nlangs
     (chsk-send! [:twitter_websockets/langs-count
{:nlangs nlangs :uid (:uid @chsk-state)}]))))
```

Codi 4.5-12 Iteració 5: funció notify-server

Cal destacar que per obtenir el valor del camp d'entrada ho fem mitjançant la funció d'Om `get-node`. Podem fer-ho així degut a que en la definició del camp d'entrada hem donat valor a l'atribut html `ref`, el qual serveix a Om per poder referenciar aquest camp de la forma que ho fem en aquest punt.

4.5.5 Aparença de la pàgina

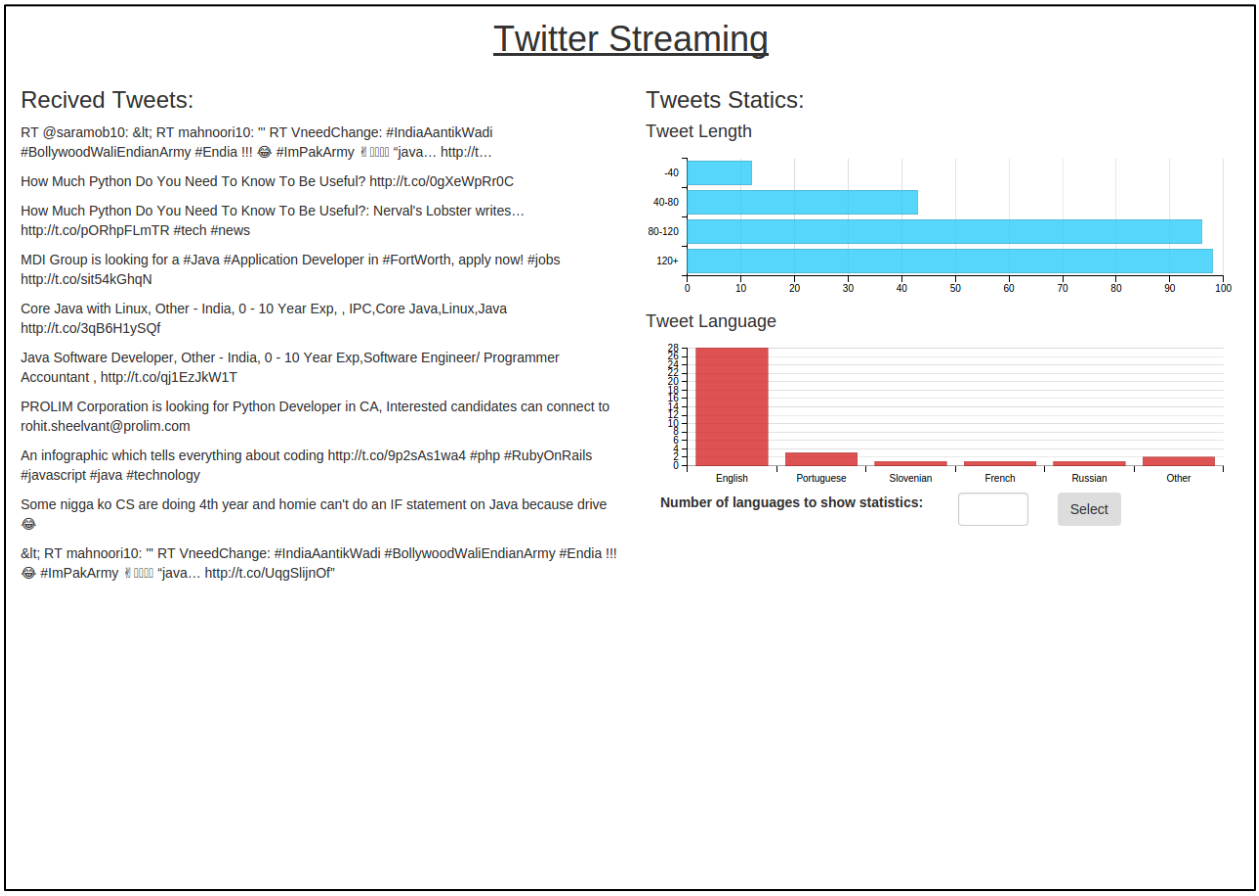


Figura 14 Aparença de la pàgina despres de la Iteració 5

Aquesta és l'aparença final de la pàgina. Podem veure que hem canviat l'orientació de les barres de la gràfica de la longitud dels *tweets*. També hem fet que la barra corresponent a l'etiqueta "Altres" surti en última posició en la gràfica dels idiomes dels *tweets*.

Finalment, hem introduït un camp d'entrada per tal de poder indicar el nombre d'idiomes dels quals volem rebre les estadístiques. En introduir un valor en aquest camp i prémer el botó "Select" es canvia el nombre de barres que té la gràfica dels idiomes. El número seleccionat es mantindrà fins a indicar-n'hi un altre. En la següent imatge podem veure la gràfica en haver seleccionat rebre estadístiques de 3 idiomes.

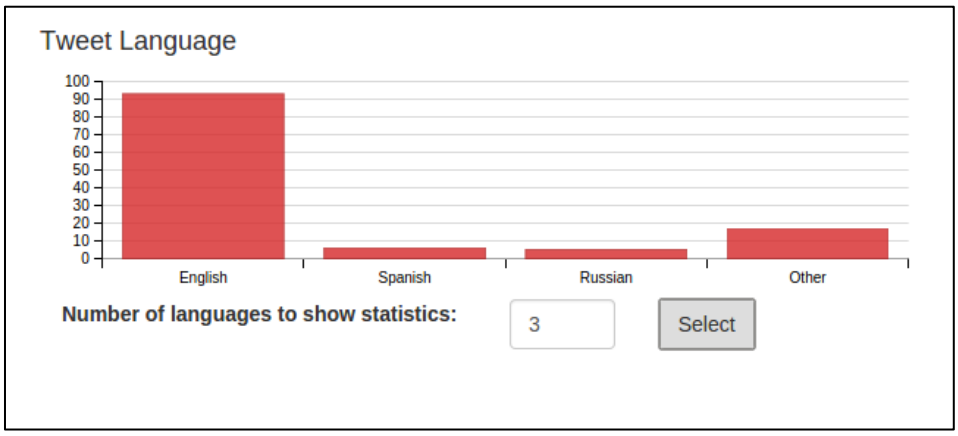


Figura 15 Gràfica dels idiomes en haver seleccionat veure només tres idiomes

Podem veure que només es mostra informació dels tres idiomes amb major nombre de *tweets* rebuts i la barra corresponent al la resta de *tweets* ha incrementat, ja que com menys idiomes vulguem mostrar, més *tweets* quedaran agrupats en la barra d'"Altres".

4.5.6 Conclusions

En aquesta iteració, el primer que hem fet ha sigut valorar si era possible implementar els requeriments 7 i 8, els quals finalment hem optat per descartar degut a la seva complexitat. És per això que hem buscat alternatives factibles en funció a la nostra temporització i que alhora ens permetessin explorar més a fons alguns aspectes dels que havíem vist fins ara.

Una vegada implementats els objectius que ens havíem marcat per a aquesta iteració podem dir que hem triat bé aquests objectius, ja que ens han permès entendre millor el funcionament de la llibreria Dimple.js i els diferents paràmetres que podem indicar a la funció que crea les gràfiques per tal de personalitzar la gràfica al nostre gust. No obstant això, ens ha quedat pendent unificar les funcions encarregades de crear cadascuna de les gràfiques, les quals són pràcticament idèntiques però en una l'eix de les X és categòric i el de les Y numèric i en l'altra és a la inversa.

A més a més, hem pogut experimentar amb el tipus de comunicació pel *websocket* que ens faltava per veure, la del client cap al servidor. Cal dir que és pràcticament idèntic a

com es fa en el sentit invers i és per això que no hem tingut cap tipus de problema en aquest sentit. El que si que ens ha causat una mica de problemes ha estat el fet d'haver de tractar en el servidor els diferents clients connectats a l'aplicació per separat, ja que pot ser que un dels clients ens hagi indicat un número d'idiomes dels quals vol rebre estadístiques i un altre client un altre número. No obstant això, també hem resolt aquest apartat amb facilitat.

Finalment, hem pogut veure la forma com es gestionen els camps d'entrada en Om, ja que és diferent de com es fa habitualment.

5 MILLORES I TREBALL FUTUR

En primer lloc, creiem que les funcionalitats que hem implementat en el treball han quedat totes ben acabades visual i funcionalment. No obstant això, hi ha un punt on podríem introduir una millora i és en la gràfica dels idiomes. En algun moment durant el transcurs del treball hem pensat que estaria bé que la barra corresponent a l'etiqueta "Altres" fos d'un altre color per diferenciar-la de les que representen els diferents idiomes. Així doncs, podríem fer aquest canvi per millorar l'aparença i claredat de les dades que es mostren en aquesta gràfica. Tret d'això, les millores que podríem fer en el treball estarien encarades a la millora del codi, fent-hi alguna refactorització per millorar la claredat del codi o per implementar alguna part d'una forma més funcional.

En segon lloc, quant al treball futur que podríem realitzar, han quedat algunes funcionalitats en el tinter, que les hem deixat de banda degut a la complexitat d'aquestes i la falta de temps per implementar-les. A més a més, donat que per realitzar aquest treball hem tingut que passar períodes d'aprenentatge del llenguatge i de les diferents tecnologies que hem utilitzat, només hem acabat implementant alguns dels elements dels que constaria un `dashboard`. Per tant, el que continuaríem fent a partir d'ara seria implementar nous elements i característiques del `dashboard`.

Així doncs, la següent funcionalitat que podríem afegir a l'aplicació seria la de permetre als usuaris filtrar els *tweets* que es mostren en la pàgina. Aquests es podrien filtrar segons una o vàries paraules introduïdes en un camp d'entrada, les quals haurien d'estar contingudes en el text del *tweet* per ser mostrat. També podríem permetre que els usuaris seleccionessin una de les barres de les gràfiques per tal de filtrar els *tweets* que es mostren segons la barra seleccionada. És a dir, filtraríem per la longitud, o bé per l'idioma. Inclús podríem permetre als usuaris fer una combinació d'aquests tres filtres.

D'altra banda, un altre tret característic dels `dashboards` és que permeten veure les estadístiques en múltiples vistes. En la nostra pàgina també podríem tenir varies formes de veure les estadístiques calculades (en forma de text, gràfica de barres,

gràfica circular…). En aquest cas es permetria als usuaris canviar per unes pestanyes la forma en què es volen visualitzar les estadístiques de les dades.

Finalment, en l'apartat **Requeriments inicials** vam definir-ne dos que finalment no els hem implementat. Aquests dos són:

- El sistema permetrà seleccionar una de les franges de longitud dels *tweets* i mostrarà, en aquest cas, només la informació del país d'on provenen els *tweets* d'aquesta longitud.
- El sistema permetrà seleccionar un país i mostrarà, en aquest cas, només la informació de la longitud dels *tweets* que provenen d'aquest país.

Així doncs, no ens podem oblidar d'aquests dos requeriments per incloure'ls com a treball futur. Tot i que caldria pensar com fer-ho per implementar conjuntament les funcionalitats esmentades anteriorment i aquests requeriments, ja que podrien xocar una mica degut a que totes requereixen de la interacció de l'usuari amb les barres de les gràfiques.

A més a més, tampoc podem oblidar-nos de la idea que teníem inicialment de calcular una estadística del lloc d'on provenen els *tweets* i mostrar-la en un mapa, la qual vam descartar degut a que l'API de Twitter no ens facilitava el lloc d'on provenen els *tweets*. No obstant això, en un futur possiblement s'actualitzi l'API de Twitter per incloure aquesta informació i podrem implementar aquesta funcionalitat.

Per acabar, una última tasca que no hem fet en el nostre projecte i que caldria fer abans de continuar implementant noves funcionalitats és la d'actualitzar les versions de Clojure, ClojureScript i totes les llibreries que hem utilitzat. Com ja hem comentat en l'apartat **ClojureScript**, no hem realitzat aquesta tasca degut a que és un procés una mica costós que pot donar lloc a inestabilitat en el codi en cas de no actualitzar bé les versions. És per això que hem decidit no actualitzar-les i garantir l'estabilitat del projecte.

6 CONCLUSIONS

Una vegada acabat el treball, podem dir que hem superat de manera força satisfactòria les expectatives que teníem d'aquest. Hem aconseguit ser capaços de programar fluidament en el llenguatge de programació funcional Clojure, coneixent així un altre tipus de programació diferent del que havíem vist fins ara, que era la programació orientada a objectes. Fins i tot hem aprofundit força en alguns trets característics del llenguatge, com és el cas de les estructures de dades persistents, de les quals hem entès fins i tot la forma en què Clojure les gestiona internament.

Cal destacar que un dels mètodes utilitzats per aprendre Clojure ha estat mitjançant llibres. És una forma d'aprendre un llenguatge que mai havia experimentat i m'ha sorprès positivament. Els llenguatges que conec els he aprés mitjançant explicacions d'algú o amb tutorials. No obstant això, fent-ho d'aquestes formes el que aprenem és a programar en un llenguatge determinat inclús sense conèixer el llenguatge. En canvi, en un llibre s'expliquen alguns aspectes interns del llenguatge que difícilment coneixeríem programant en aquest i que ens faran esdevenir millors programadors del llenguatge.

D'altra banda, un altre mètode utilitzat per a aprendre Clojure ha estat agafant projectes de GitHub en aquest llenguatge i intentant entendre el codi [R2]. Aquest mètode també ajuda molt a entendre el llenguatge degut a que ens obliga a pensar com es comporta cada tros de codi. Cal destacar que per utilitzar aquest mètode per aprendre un llenguatge és de gran ajuda tenir algú que entengui perfectament el llenguatge i ens pugui resoldre dubtes que ens sorgeixin i no siguem capaços de resoldre nosaltres. En el nostre cas aquesta persona ha sigut el tutor del projecte, el Juan Manuel Gimeno.

A més a més, també hem aprofundit força en les idees que presenta Om. En aquest cas no hem llegit cap llibre, ja que és una llibreria relativament nova i encara no hi ha tanta documentació com tenim de Clojure. No obstant això, ens hem ajudat de diversos articles i presentacions que ens han fet entendre les idees que s'amaguen darrere de la llibreria. La diferència amb els llibres és que en aquest cas els articles es centren en

una part més acotada de la tecnologia que expliquen. La qual cosa fa que necessitem més d'un article per conèixer les diferents característiques del que estem aprenent.

Finalment, també hem treballat amb la llibreria de JavaScript Dimple.js. En aquest cas aquesta llibreria no presentava unes idees tan innovadores com en el cas d'Om. És per aquest motiu que ens hem limitat més a utilitzar la llibreria per tal de crear les gràfiques que volíem, deixant una mica de banda la forma com ho fa per crear les gràfiques.

Respecte a les diferents iteracions, no hem tingut excessius problemes per a la seva implementació i, excepte en algun punt que ens hem quedat encallats, les hem solucionat amb relativa facilitat i rapidesa. No obstant això, el que si que hem hagut de fer en nombroses ocasions ha estat fer refactoritzacions del codi que implementàvem inicialment per a resoldre les iteracions. Això ha estat degut a que estàvem acostumats a programar de forma imperativa i en Clojure, tot i ser un llenguatge de programació funcional, també podem resoldre alguns problemes de forma imperativa. Per tant, refactoritzavem el codi per implementar-ho tot de la forma més funcional possible. A més a més, algunes de les refactoritzacions que hem fet han estat a causa que desconeixíem algunes funcions de Clojure que ens facilitaven la resolució del problema el qual estàvem tractant. Cal destacar que tots aquests errors han sigut determinants per conèixer millor el llenguatge i aprendre a programar funcionalment en aquest. En la següent gràfica generada automàticament per GitHub es mostra una comparativa entre les línies de codi afegides (en verd) i eliminades (en roig) en relació amb el temps. En ella podem observar com la quantitat de línies de codi eliminades ha augmentat notablement en l'última fase del projecte degut a les refactoritzacions esmentades. D'altra banda, pot sorprendre la gran quantitat de línies que indica que es van afegir en començar el projecte. No obstant això, aquesta dada és enganyosa degut a que en el primer `commit` vam pujar molts arxius de configuració d'IntelliJ, a part de tot el codi que ens va crear automàticament la plantilla "chestnut".

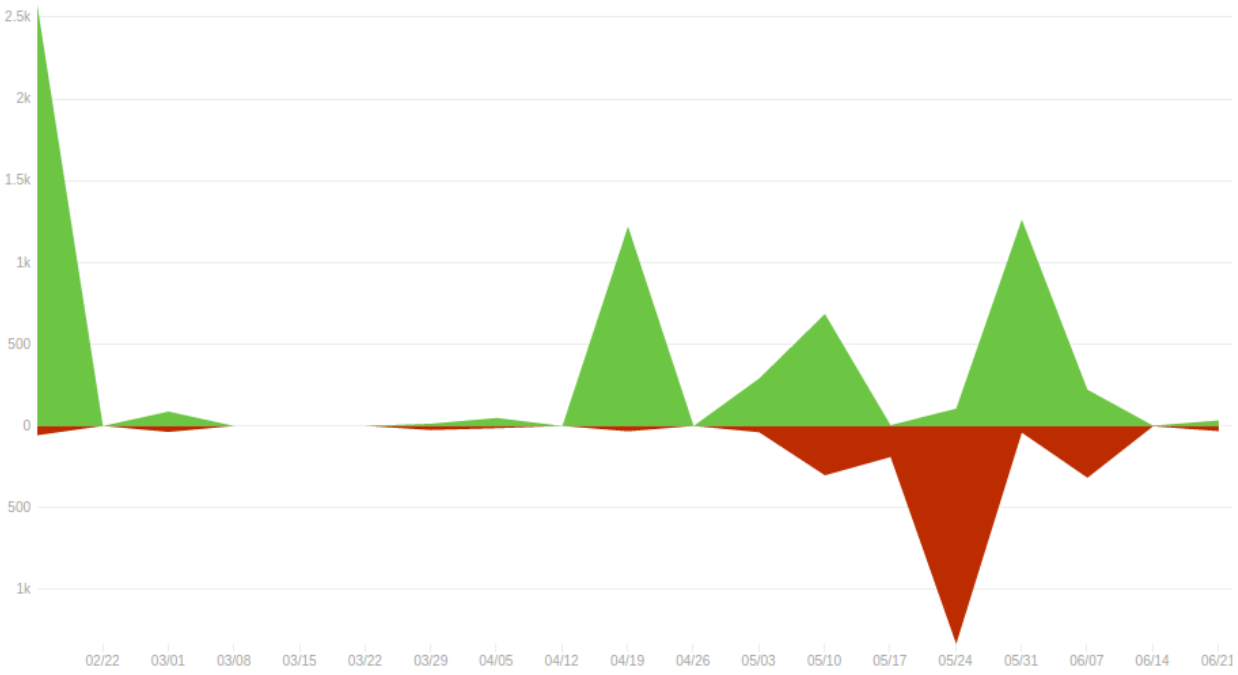


Figura 16 Gràfica que mostra les línies de codi afegides (en verd) i eliminades (en roig) en el repositori GitHub al llarg del temps

Finalment, en aquest punt també cal fer una comparativa entre la planificació temporal que vam fer en l'apartat i la temporització real del projecte. Per veure aquesta temporització també ens ajudarem d'una de les gràfiques que autogenera GitHub, en la qual es mostren el nombre de `commits` realitzats per setmanes.

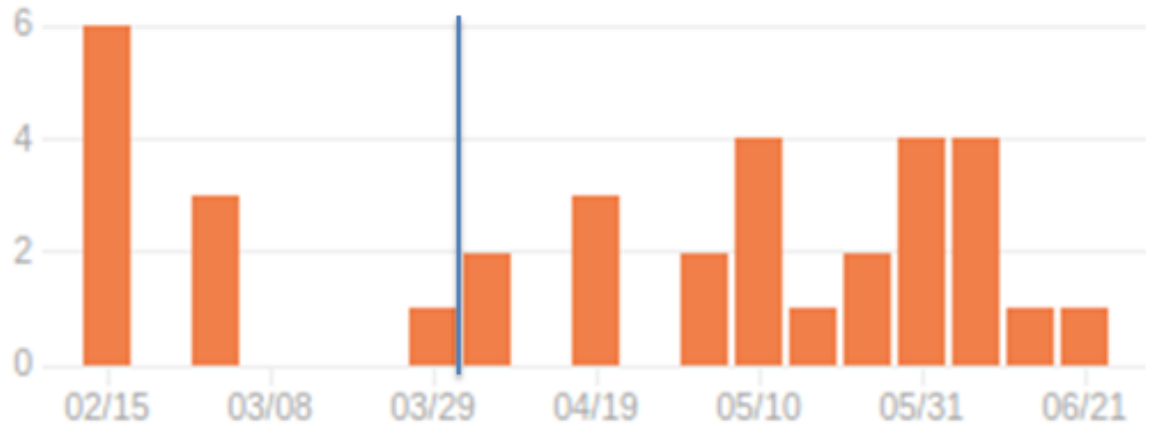


Figura 17 Gràfica que mostra els commits fets al repositori GitHub per setmanes

Podem observar que a partir del moment en que vam fer la planificació temporal de les iteracions, el qual s'indica en la gràfica mitjançant la línia blava, hem fet `commits` com a mínim cada dos setmanes. A més a més, no hem necessitat més de dues setmanes per a realitzar cadascuna de les iteracions, a excepció de la quarta, la qual ens va portar aproximadament quatre setmanes degut a que en aquesta iteració vam començar a treballar amb la llibreria Dimple.js, la qual no coneixíem i vam tenir que fer una mica d'aprenentatge. No obstant això, en la planificació temporal que vam fer vam comptar una sisena iteració que finalment no hem fet, amb la qual cosa recuperem aquestes dues setmanes de més que ens ha ocupat la iteració 4. D'altra banda, les iteracions 4 i 5 les hem implementat finalment en una única iteració però la cinquena iteració que hem realitzat no estava contemplada en la planificació temporal. Per tant, ambdós casos es compensen en quant a temporització. Així doncs, podem dir que ens hem cenyit força a la planificació temporal que havíem fet.

7 BIBLIOGRAFIA

7.1.1 Llibres

[1] Butcher, P. (2014). Seven concurrency models in seven weeks. When threads unravel. The Pragmatic Programmers, LLC.

[2] Halloway, S. (2009). Programming Clojure. The Pragmatic Programmers, LLC.

[3] Higginbotham, D. (en progrés). Clojure for the Brave and True. Disponible a: http://www.braveClojure.com/.

[4] Nehlsen, M. (en progrés). Building a system in Clojure (and ClojureScript). Leanpub.

[5] Sierra, S., VanderHart, L. (2012). ClojureScript: Up and Running. Functional Programming for the Web. O'REILLY.

[6] Sotnikov, D. (2014). Web Development with Clojure. Build Bulletproof Web Apps with Less Code. The Pragmatic Programmers, LLC.

7.1.2 Presentacions

[P1] Gimeno, J. M. (2010). Functional Programming in Clojure. Disponible a: http://es.slideshare.net/jmgimeno/functional-programming-in-Clojure.

[P2] Halloway, S. (2014). Simple by Design: Clojure. Disponible a: http://es.slideshare.net/AllThingsOpen/Clojure-simplebydesign.

[P3] Hickey, R. (2009). Persistent Data Structures and Managed References. Disponible a: http://es.slideshare.net/michael.galpin/persistent-data-structures-and-managed-references.

[P4] Pawlicka, A. (2014). Om nom nom nom. Disponible a: http://www.slideshare.net/annapawlicka/Om-nom-nom-nom.

[P5] Pawlicka, A. (2014). Reactive data visualizations with Om. Disponible a: http://www.slideshare.net/annapawlicka/reactive-data-visualisations-with-Om.

7.1.3 Articles

[A1] Antukh, A. (2015). ClojureScript Tutorial. Disponible a: https://www.niwi.nz/cljs-workshop/.

[A2] Garza, B. (2014). Thoughts on Clojure and how to learn it. Disponible a: https://medium.com/@bryangarza/thoughts-on-Clojure-and-how-to-learn-it-685844b4e94b.

[A3] Hickey, R. (2012). Reducers - A Library and Model for Collection Processing. Disponible a: http://Clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html.

[A4] Rumford, I. (2014). Some trivial examples of using Clojure Transducers. Disponible a: http://ianrumford.GitHub.io/blog/2014/08/08/Some-trivial-examples-of-using-Clojure-Transducers/.

[A5] L'orange, J. N. Understanding Clojure's Persistent Vectors, pt. 1. Disponible a: http://hypirion.com/musings/understanding-persistent-vector-pt-1.

[A6] Schuck, P. (2014). Transducers: Clojure's next big idea. Disponible a: http://bendyworks.com/transducers-Clojures-next-big-idea/.

[A7] Shira, E. (2014). The Trifecta of ClojureScript, Om and core.async. Disponible a: http://elbenshira.com/blog/trifecta-ClojureScript-Om-coreasync/.

[A8] Zhlobich, A. (2015). Clojure. Transducers, Reducers and Other Stuff. Disponible a: http://kukuruku.co/hub/funcprog/Clojure-transducers-reducers-and-other-stuff.

7.1.4 Repositoris GitHub

[R1] Gimeno, J. M. Om Twitter. <https://GitHub.com/jmgimeno/Om-twitter> [Darrera consulta: 1 de març de 2015]

[R2] Gimeno, J. M. Post to screen. <https://GitHub.com/jmgimeno/post-to-screen> [Darrera consulta: 31 de maig de 2015]

[R3] Miller, A. Core.async. <https://github.com/clojure/core.async> [Darrera consulta: 29 d'abril de 2015]

[R4] Munen, A. Om Basic Tutorial. https://github.com/omcljs/om/wiki/Basic-Tutorial [Darrera consulta: 13 de juny de 2015]

[R5] Nehlsen, M. BirdWatch. <https://GitHub.com/matthiasn/BirdWatch> [Darrera consulta: 19 de febrer de 2015]

[R6] Nolen, D. Om Wiki. <https://GitHub.com/omcljs/Om/wiki> [Darrera consulta: 13 de juny de 2015]

[R7] Pawlicka, A. Pumpkin. <https://GitHub.com/annapawlicka/pumpkin> [Darrera consulta: 10 de maig de 2015]

[R8] Pawlicka, A. Om-data-vis. <https://GitHub.com/annapawlicka/om-data-vis> [Darrera consulta: 10 de maig de 2015]

[R9] Ptoussanis, P. Sente, channel sockets for Clojure. <https://GitHub.com/ptaoussanis/sente> [Darrera consulta: 30 de maig de 2015]

7.1.5 Pàgines web

[W1] 4Clojure. <http://www.4Clojure.com/problems> [Darrera consulta: 7 de maig de 2015]

[W2] Clojure Cheatsheet. <http://Clojure.org/cheatsheet> [Darrera consulta: 13 de juny de 2015]

[W3] ClojureScript Cheatsheet. <http://cljs.info/cheatsheet/> [Darrera consulta: 13 de juny de 2015]

[W4] Cursive Clojure. <https://cursiveClojure.com/> [Darrera consulta: 10 de maig de 2015]

[W5] Dimple.js. <http://dimplejs.org/> [Darrera consulta: 2 de juny de 2015]

[W6] React. <https://facebook.GitHub.io/react/index.html> [Darrera consulta: 7 d'abril de 2015]

[W7] Scalable Vector Graphics (SVG). <http://www.w3.org/Graphics/SVG/> [Darrera consulta: 10 de maig de 2015]

[W8] Twitter, Inc. Twitter API Overview. <https://dev.twitter.com/overview/api> [Darrera consulta: 25 de febrer de 2015]